UNIVERSIDAD AUTONOMA DE MADRID

ESCUELA POLITECNICA SUPERIOR

Doble Grado en Ingeniería Informática y Matemáticas

TRABAJO FIN DE GRADO

ESTUDIO Y ANÁLISIS DE UN PROTOCOLO CRIPTOGRÁFICO DE ANONIMATO Y PRIVACIDAD JUSTA

Autor: Omar Molinero Gil
Tutor: Francisco de Borja Rodríguez Ortiz

Junio 2018

ESTUDIO Y ANÁLISIS DE UN PROTOCOLO CRIPTOGRÁFICO DE ANONIMATO Y SEGURIDAD JUSTA

AUTOR: Omar Molinero Gil
TUTOR: Francisco de Borja Rodríguez Ortiz

Grupo de Neurocomputación Biológica (GBN)
Dpto. de Ingeniería Informática
Escuela Politécnica Superior
Universidad Autónoma de Madrid
Junio de 2018

Resumen (castellano)

La privacidad y anonimia en la red es un gran problema al que nos enfrentamos en la actualidad. Actualmente, hay una cantidad masiva de datos personales que circulan por la red. Esto ha llevado a que los servicios de internet que manejan principalmente estos datos tengan que regularse a través de unas leyes con las cuales los usuarios puedan ejercer su derecho a su confidencialidad.

Sin embargo, existen soluciones criptográficas para evitar que se propague la mayor parte de la información individual no deseada. Mediante firmas grupales, un usuario consigue identificarse como miembro de un grupo, y podría realizar acciones sin tener que revelar su verdadera identidad personal. Esto genera un grave problema para el sistema si el usuario realiza acciones ilícitas y no se puede averiguar quién es. Por esta razón, estas firmas grupales suelen ser revocables, para poder ofrecer al servicio la identidad real de los clientes, únicamente en el caso de que estos actúen de manera incorrecta.

Partiendo de esta idea, en este trabajo de fin de grado se pretende conseguir, mediante un protocolo criptográfico, garantizar la anonimia a los usuarios de un determinado servicio. Pero en el caso de que cometan alguna acción no lícita, el sistema podrá revocar el anonimato a los usuarios implicados, sin afectar ni al rendimiento de este ni al resto de clientes.

Para ello partimos de un sistema en el que se distinguen cuatro actores. Uno de ellos actuará como el cliente del servicio y los otros tres proporcionarán anonimia gracias a la división de información. Los tres actores del servidor se dividirán la información del cliente mediante un protocolo con cuatro fases. En la primera fase, un actor se encarga de gestionar la información real del cliente y asignarle un identificador como pseudónimo en el servicio. En la segunda fase, otro actor se encarga de proporcionar claves únicas y personales a cada cliente que se identifique correctamente con su pseudónimo. En la tercera fase, el último actor del sistema proporcionará el servicio a los clientes que posean las claves correctas. Además, en la cuarta fase, este último actor podrá intercambiar información con los otros dos actores en el caso de que sea necesario revocar la anonimia a un cliente.

En la última parte de este trabajo, se estudiará y analizará los resultados de este protocolo criptográfico, así como el nivel de anónima alcanzado. Partiendo de la hipótesis de que los actores actúan de la manera correcta según el protocolo, la conexión del cliente con el proveedor del servicio (que denominaremos en este trabajo: manager del servicio) se realiza de manera completamente anónima. Pero durante todo el proceso en el que el cliente consigue el acceso anónimo, ha tenido que proveer a los otros actores del sistema la información necesaria para poder revocar su anonimato en caso de que sea necesario.

Palabras clave (castellano)

Anonimia, privacidad, criptografía, función hash, firma digital, anonimato justo, revocabilidad, trazabilidad, conjunto de anonimia, pseudoanonimia, prueba de conocimiento cero, división de información, protocolo criptográfico, seguridad informática, contratos inteligentes

Abstract (English)

Privacy and anonymity in the network is a big problem that we face today. Currently, there is a massive amount of personal data circulating on the network. This has led to the Internet services that handle these data mainly have to be regulated through laws with which users can exercise their right to confidentiality.

However, there are cryptographic solutions to prevent the circulation of most of the unwanted individual information. Through group signatures, a user is able to identify himself as a member of a group, and could perform actions without having to reveal his true personal identity. This generates a serious problem for the system if the user performs illicit actions and the service can not find out who is. For this reason, these group signatures are usually revocable, in order to offer the service the real identity of the clients, but only in the case that they act in an incorrect way.

Starting from this idea, in this end-of-degree project, it is intended to guarantee, through a cryptographic protocol, the anonymity to the users of a certain service. But in the case that they commit any unlawful action, the system may revoke the anonymity of the users involved, without affecting the performance of this or the other clients.

We start from a system in which four actors are distinguished. One of them will act as the customer of the service and the other three will provide anonymity thanks to the information division. The three actors of the server will divide the information of the client through a protocol with four phases. In the first phase, an actor is responsible for managing the real information of the client and assigning an identifier as a pseudonym in the service. In the second phase, another actor is responsible for providing unique and personal keys to each client who correctly identifies with his pseudonym. In the third phase, the last actor of the system will provide the service to the clients that have the correct keys. In addition, in the fourth phase, this last actor may exchange information with the other two actors in case it is necessary to revoke the anonymity of a client.

In the last part of this thesis, we will study and analyze the results of this cryptographic protocol, as well as the level of anonymity reached. Accepting the hypothesis that the actors act in the correct way according to the protocol, the client's connection with the service provider (which we will call in this paper: service manager) is done completely anonymously. But during the whole process in which the client obtains anonymous access, he has had to provide the other actors of the system with the necessary information to be able to revoke his anonymity.

Keywords (inglés)

Anonymity, privacy, cryptography, hash function, digital signature, fair anonymity, revocability, traceability, anonymity set, pseudoanonymy, zero knowledge proof, information division, cryptographic protocol, informatic security, smart contracts

Agradecimientos

En primer lugar, me gustaría agradecer en este trabajo a D. Francisco de Borja Rodríguez por toda la ayuda y paciencia que ha tenido conmigo para conseguir hacer este TFG. De la misma forma, quisiera agradecer la ayuda a Álvaro Marcos por todas las sesiones de brainstorming que tuvimos juntos hasta llegar al resultado mostrado en este TFG.

Por otra parte, quiero agradecer a mis compañeros y amigos de la carrera, los cuales nos hemos estado apoyado mutuamente en este largo camino del doble grado.

Para finalizar, también me gustaría agradecer a mi familia por todo su esfuerzo ya que me han ayudado mucho en llegar hasta donde estoy hoy en día.

INDICE DE CONTENIDOS

INDICE DE FIGURAS

INDICE DE TABLAS

1 Introducción

1.1 *Motivación*

La privacidad y anonimia en la red es un problema complicado al que nos enfrentamos en la actualidad. Recientemente podemos leer noticias sobre el robo masivo de datos que sufrió la red social Facebook a principios de 2018. Entonces, ¿Cómo de seguros son nuestros datos en internet? ¿Qué es exactamente lo que cada servicio de internet conoce y guarda sobre mí y qué puede hacer con esos datos? ¿Por qué necesito identificarme para consumir ciertos servicios en los que importe la acción (por ejemplo, comprar algo) y no la identificación del cliente (saber exactamente el individuo que lo compra)?

La solución a este problema no es sencilla. Existen distintas herramientas con las que se puede conseguir seguridad durante el intercambio de información en la red entre el cliente y el servidor. Mediante la criptografía se puede conseguir ocultar la información enviada (confidencialidad de la información). Las funciones hash ayudan a asegurar al receptor que la información enviada por el emisor no ha sido modificada (integridad de la información). Las firmas digitales ayudan a garantizar que realmente el emisor de un determinado mensaje es quien dice ser (autenticación de la información). Pero esta última parte hace que el emisor pierda su anonimia al tener que aportar una prueba mediante su firma de que en realidad es él el origen del mensaje y no una tercera persona que quiere hacerse pasar por él. Como solución a esto, existen esquemas de firmas digitales más complejas como podrían ser las firmas grupales. En este tipo de firmas, los emisores se identificarán como pertenecientes a un grupo válido, el cual se llamará conjunto de anonimia. El receptor sabrá que el origen del mensaje corresponde a un miembro de dicho conjunto, pero no podrá identificar al individuo exactamente.

Pero aún quedan múltiples problemas que se mantienen. Respecto a la privacidad, aun garantizando el envío seguro de la información, el cliente no puede saber si el servicio hace un uso abusivo o comercial con su información personal. Por ello, las leyes que regulan la privacidad en la red están siendo revisadas y actualizadas, como ha ocurrido con la implantación de las nuevas leyes de privacidad europeas en mayo de 2018. Respecto a la anonimia, existen servicios (por ejemplo, Facebook), a los cuales es necesario la identificación del cliente para poder consumir el servicio (en el caso de Facebook, acceder a su perfil). Pero en el caso de los servicios que permitan la anonimia de acceso, el mayor problema es su incapacidad de detectar al culpable de realizar acciones ilegales contra el sistema si se da el caso en el que todos los usuarios accedieron de manera completamente anónima. Por esta razón la anonimia total en internet no es viable, y si se quiere pensar en un sistema que proporcione a los clientes un cierto nivel de anonimia, se debe de tener en cuenta la capacidad del sistema para poder identificar con precisión a los clientes en el caso que sea necesario. Es lo que se conoce como revocación de identidad.

1.2 *Objetivos*

El objetivo principal de este trabajo es la capacidad de otorgar un anonimato justo a los usuarios válidos de una determinada aplicación o web. Dicho anonimato únicamente podrá ser revocado en el caso de que cometan alguna infracción.

Con el fin de alcanzar el objetivo principal, podemos destacar los siguientes objetivos secundarios:

- Profundizar y ampliar conocimientos sobre seguridad informática.
- Abordar el actual problema del anonimato en internet.
- Implementar un protocolo que consiga el correcto acceso a una aplicación básica sin la necesidad por parte del usuario de desvelar su identidad real.
- Estudiar y analizar las posibles formas de ataques y vulnerabilidades del sistema diseñado, además de debatir la posibilidad o no de añadir mejoras para protegerse de estas debilidades.

1.3 *Organización de la memoria*

La estructura de la memoria se divide en los siguientes capítulos:

- **Estado del arte:** En este capítulo se muestran los conceptos teóricos más importantes tratados en este TFG. Empieza con una breve introducción a la criptografía hasta llegar a la criptografía moderna. Le siguen nociones importantes como las funciones resumen o funciones hash y las firmas digitales. Todo esto sirve como base para tratar el problema principal del TFG que es el tema de la privacidad y el anonimato, en el cual se explicará el objetivo final del trabajo que es llegar a un estado de anonimato justo. Para finalizar esta sección se darán unas pequeñas pinceladas a modo resumen del tema legislativo detrás de la anonimia en internet.
- **Análisis y Diseño:** Este capítulo comenzará explicando la estructura del sistema y los actores que la forman. Más adelante, se analizarán las fases del protocolo implementado además de una breve descripción sobre el diseño de las actividades implementadas en la aplicación Android que actuará como cliente en el sistema. Para concluir, se comentará brevemente la estructura de los mensajes utilizados para la comunicación entre los distintos actores.
- **Desarrollo e integración:** Se explicará la parte técnica usada para el desarrollo, además de información sobre el software implementado y las tecnologías informáticas utilizadas.
- **Pruebas, resultados y estudio:** Este capítulo sirve para mostrar el funcionamiento del sistema, así como realizar un estudio sobre el tiempo de respuesta y el nivel de anonimia y seguridad alcanzado.
- **Conclusiones y trabajo futuro:** Esta sección tratará sobre las conclusiones que deja el trabajo y los posibles avances que podrán realizarse.
- **Anexos:** Se incluirá información adicional detallada sobre conceptos y código utilizados en este TFG.

2 Estado del arte

En este capítulo explicaremos los distintos conceptos utilizados en este trabajo. Comenzaremos hablando sobre la criptografía y los métodos de cifrado modernos. También hablaremos sobre las funciones hash y las firmas digitales. Todo esto servirá como base para explicar el problema principal relacionado con la anonimia y privacidad, y añadiremos un breve resumen sobre las leyes de protección de datos actuales. Para concluir este capítulo hablaremos de dos tecnologías de la información que tendrán una gran importancia en este trabajo: los contratos inteligentes y la cadena de bloques (Blockchain).

2.1 *Criptografía*

La criptografía (del griego "*criptos*" oculto y "*grafé*" escritura) se define como el estudio de algoritmos, sistemas y protocolos usados con el fin de ocultar la información. Además, hoy en día también es usada para otorgar seguridad a las comunicaciones y a las entidades que se comunican.

El concepto y el uso de la criptografía por parte del hombre ha ido evolucionando a lo largo de la historia, como podemos observar en el anexo C. El resultado de esta evolución son los métodos de cifrado moderno que utilizamos hoy en día para conseguir seguridad en internet.

Estos métodos de cifrado moderno se basan en una serie de principios que Auguste Kerckhoffs definió en 1883 [1]:

i. Si el sistema no es teóricamente irrompible, al menos tiene que serlo en la práctica.
ii. La efectividad del sistema no debe depender de que su diseño permanezca en secreto.
iii. La clave debe ser fácilmente memorizable de manera que no haya que recurrir a notas escritas.
iv. Los criptogramas deberán dar resultados alfanuméricos.
v. El sistema debe ser operable por una única persona.
vi. El sistema debe ser fácil de utilizar.

En particular, el principio más deseado es el ii, el cual se conoce como principio de Kerckhoffs por antonomasia.
Podemos dividir los métodos de cifrado moderno según la forma en compartir las claves.

2.1.1 Cifrado simétrico moderno

En el cifrado simétrico, las entidades que se comunican utilizan la misma clave tanto para encriptar como para desencriptar los mensajes.
Podemos encontrar el primer gran inconveniente que es el previo intercambio de claves necesario para que tanto el emisor como el receptor puedan comunicarse de forma segura.
Otro inconveniente es el número de claves que se necesitan para que n entidades se comuniquen de forma segura. Está claro que la misma clave para todos no vale porque entonces todas las entidades tendrían acceso a las conversaciones ajenas. Es necesario que cada par de nodos en una conversación posean una clave única para ese intercambio de información, lo que eleva el número de claves totales a $\frac{n*(n-1)}{2}$.

Los métodos de cifrado simétrico se dividen en dos dependiendo de cómo cifran el mensaje:

- Cifrado por bloque: Dividen en mensaje en bloques de k bits y los van cifrando. Ejemplos: AES (anexo G), DES, Blowfish, IDEA.
- Cifrado de flujo: Cifran el mensaje con correspondencia bit a bit sobre el flujo. Ejemplos: RC4, RC6.

2.1.2 Cifrado asimétrico moderno

El cifrado asimétrico consta de un par de claves distintas par el envío de un mensaje. Una de ellas es pública y puede ser conocida por cualquier persona, mientras que la otra es privada y únicamente la debe conocer únicamente su propietario. Cuando el emisor decide enviar un mensaje, este la cifra con la clave pública del receptor para que únicamente él pueda descifrarlo usando su clave privada; es decir, otorga confidencialidad al mensaje.

Sin embargo, si el emisor utiliza su clave privada para cifrar el mensaje, dicho mensaje perderá confidencialidad ya que cualquiera lo puede descifrar con la clave pública del emisor. Pero otorga otra característica diferente que será el objetivo de las firmas digitales como veremos en la sección 2.3, la autenticidad del emisor. Nadie más que el emisor del mensaje posee su clave privada, así que el receptor puede identificarle y además comprueba que nadie ha podido alterar el mensaje.

Este método de cifrado soluciona los problemas de intercambio de claves y del gran número de claves que se necesitaban en el cifrado simétrico, pero también tiene sus inconvenientes. Se necesita un mayor tiempo de proceso que el cifrado simétrico, sus claves suelen ser más grandes y el mensaje cifrado ocupa un mayor tamaño que el mensaje original.

Ejemplos de algoritmos que usan la criptografía asimétrica son RSA (anexo H), DSA, Diffie-Helman (anexo I) y ElGamal.
También existen tecnologías como las curvas elípticas (anexo L) que es una variante basada en las matemáticas de las curvas elípticas. Sus autores aseguran que reducen el tiempo de procesamiento y el tamaño de claves.

2.2 *Funciones hash criptográfica*

Las funciones hash criptográficas son funciones de una sola dirección (anexo K) que comprimen la entrada a una salida de menor longitud y fácil de calcular. También se las suele llamar funciones resumen.
La idea principal es encontrar una función h tal que, sea muy fácil calcular h(x)=y, pero muy difícil encontrar x conociendo únicamente y. También se necesita que sea difícil encontrar un par (x,y) de entradas que conduzcan a la misma salida h(x)=h(y). Esto es a lo que se denomina colisión.

Propiedades de las funciones hash

- Bajo coste: costes de computación pequeños para poder ser usadas en la práctica.
- Deterministas: producen siempre la misma salida ante la misma entrada.
- Uniforme: es igualmente probable que una entrada tenga una salida determinada.
- Efecto avalancha: pocos bits distintos en la entrada producen salidas muy distintas.

Usos de las funciones hash

- MAC: permiten comprobar la autenticidad e integridad del mensaje.
- MDC: permiten detectar si han ocurrido modificaciones en el mensaje.

Ejemplos de funciones resumen son MD5 y SHA-256 (anexo M). A continuación, mostramos la Tabla 2-1 en la cual se comparan varias funciones resumen:

Algoritmo	Salida (bits)	Bloque (bits)	Tamaño máximo mensaje (bits)	Longitud palabra (bits)	Iteraciones	Colisiones encontradas	Rendimiento (MiB/s)
MD5	128	512	$2^{64} - 1$	32	64	SI	335
SHA-0	160	512	$2^{64} - 1$	32	80	SI	-
SHA-1	160	512	$2^{64} - 1$	32	80	SI	192
SHA-256	256	512	$2^{64} - 1$	32	64	NO	139
SHA-512	512	1024	$2^{128} - 1$	64	80	NO	154
SHA-3	224/256/ 384/512	1600	Ilimitado	64	24	NO	-

Tabla 2-1: Comparación funciones hash

En este trabajo se ha decidido usar como función hash la función SHA-256 debido a las propiedades que tiene son suficientes para realizar la funcionalidad requerida del sistema que se mostrará en la sección 3.2.

2.3 *Firmas digitales*

Las firmas digitales son un mecanismo criptográfico usado para identificar a la entidad originar emisora del mensaje. Es lo que se conoce como autentificación de origen y no repudio. Además, garantiza que el mensaje no ha sido alterado por un tercero, es decir, la integridad.

Podemos dividir las firmas digitales en dos:

- **Firma digital con árbitro:** Si dos usuarios quieren intercambiar información, pero desconfían entre ellos, necesitan una tercera entidad, un árbitro en los que ambos pueden confiar. El mayor inconveniente de este caso es que toda la información tiene que pasar por el árbitro.
- **Firma digital sin árbitro:** El usuario envía directamente la firma al usuario, sin necesidad de trasmitir toda la información por un árbitro. Este es el caso usado en este trabajo.

Como hemos visto previamente, mediante el cifrado con clave privada en un método de cifrado asimétrico se consigue autentificación, no repudio e integridad, pero el mensaje pierde la confidencialidad al poder ser descifrado por cualquier otra persona. Para conseguir también confidencialidad se introducen esquemas de cifrado como, por ejemplo, mediante una firma RSA, cuyo esquema se muestra en la Figura 2-1.

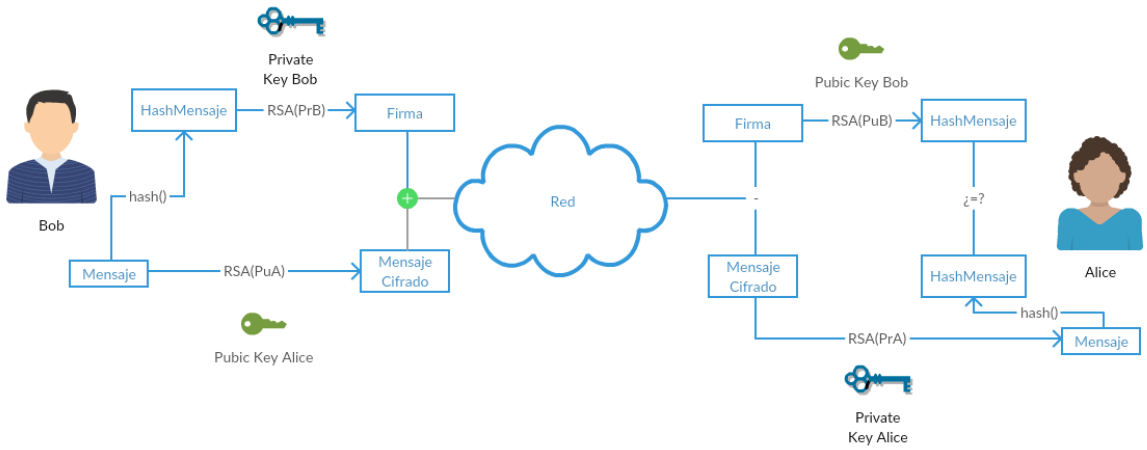

Figura 2-1: Firma digital RSA

En este esquema, Bob quiere enviar un mensaje a Alice. Para ello cifra el mensaje con la clave pública de Alice, para que sólo ella pueda descifrarlo. Además, realiza el resumen del mensaje mediante una función hash que previamente había acordado con Alice. Este resumen lo firma, cifrando con su clave privada y envía a Alice la concatenación del mensaje cifrado y la firma.
Alice recibe el mensaje y lo divide en las dos partes. Comienza descifrando el mensaje cifrado con su clave pública. Si el emisor ha sido Bob y el mensaje no ha sido alterado, el hash del mensaje descifrado tiene que ser igual a la firma descifrada con la clave pública de Bob. Si lo son Alice puede asegurar que Bob ha sido el emisor, ya que ninguna otra persona ha podido firmar como Bob porque nadie más que el posee su clave privada.
Este es el esquema que será el utilizado durante este trabajo para firmar los mensajes.

2.4 *Firmas digitales avanzadas*

Además de las firmas digitales explicadas en la sección anterior, podemos mencionar otro tipo de firmas más avanzadas cuyo objetivo pasa de ser el de identificar al emisor de un mensaje de manera unívoca, sino que se identifica como parte de un conjunto. Son las denominadas firmas grupales.

También podemos comentar otro tipo de firmas que son las firmas ciegas, en las que se firman unos datos sin llegar a conocerlos.

2.4.1 Firmas grupales

Este tipo de firmas se basa en la idea de las pruebas de conocimiento cero (anexo N). Estas pruebas son una serie de protocolos usados por un usuario para demostrar que posee la solución de un problema, mostrando únicamente la veracidad del conocimiento de esta solución. Esta clase de pruebas pueden ser usadas para demostrar la pertenencia de un usuario a un grupo sin la necesidad de este de identificarse.

Una firma grupal es un esquema de firma que realizan un conjunto de personas, al que denominaremos grupo. Este esquema sigue las siguientes propiedades [3]:

- Solo los mismos del grupo pueden firmar mensajes.
- El receptor de la firma puede verificar la validez de esta, pero no la identidad del emisor.
- En el caso de disputa, la firma puede ser "abierta". Esto revela la identidad real del firmante.

Además, existen cierto tipo de sistemas de firmas grupales en los que se podrá revocar a los usuarios para que no sean capaces de volver a firmar dentro del grupo. En el anexo O mostraremos unos ejemplos concretos de este caso.

2.4.2 Firmas digitales ciegas

La idea de este tipo de firmas es conseguir una serie de datos firmados cuyo contenido no es conocido por el usuario firmante. Es decir, son protocolos entre dos usuarios (U y V) en los que U (el firmante) firma digitalmente una serie de datos enviados por V sin llegar a conocer dichos datos.

Para ello se necesita:

a. Un protocolo de firma digital diseñado por U. Llamamos S(m) a la firma del mensaje m.
b. Dos funciones *f* y *g* que únicamente conoce V. *f* se llama a la función de ocultación y *g* a la función de recuperación. Estas funciones cumplen $g(S(f(m))) = S(m)$.

Proceso:

i. V crea un hash de su mensaje.
ii. V ciega dicho hash "$f(m)$" y se lo envía a U.
iii. U realiza la firma "$S(f(m))$" y se lo envía a V.
iv. V recupera la firma usando la función de recuperación $g(S(f(m))) = S(m)$.

Las aplicaciones de las firmas digitales ciegas se pueden encontrar más comúnmente en esquemas de voto electrónico y en esquemas para dinero electrónico.

2.5 *Privacidad y Anonimato*

La privacidad y el anonimato es un problema actual en la red difícil de tratar. Muchos servicios de internet, como podría ser Amazon o Facebook, poseen mucha información de nuestra parte, la cual puede ser usada por terceros para conseguir beneficios. Es muy común que después de hacer una compra (o simplemente una búsqueda) a través de internet nos aborden con publicidad sobre el mismo tema. Entonces, ¿dónde queda el anonimato del usuario? ¿Es necesario identificarse para realizar todas las acciones en internet? ¿Y qué pasa con toda la información que las empresas tienen guardado sobre cada usuario? ¿Cómo el usuario puede estar seguro de que no acaba en otras manos?

2.5.1 Privacidad

La privacidad consiste en que la información que han compartido dos entidades solamente pueda ser conocido por ellas.

La privacidad a nivel de comunicación se consigue gracias al cifrado. Como hemos visto en la sección 2.1, se puede garantizar la confidencialidad del mensaje a través del cifrado simétrico o asimétrico.

En el cifrado simétrico (como por ejemplo AES), ya hemos explicado que es necesario un previo intercambio de claves a través de una red no segura. Esto quiere decir que no se puede llegar a una clave en común sin más porque cualquier otra persona podría ver también esa clave y la conversación dejaría de ser privada.
Para garantizar que los usuarios puedan intercambiar la clave de manera segura, existen protocolos/sistemas de clave pública como Diffie-Helman o RSA.
En el cifrado asimétrico, es importante que cuando se cifra un mensaje con la clave pública del receptor, dicha clave no haya sido cambiada intencionadamente por otra persona, ya que entonces, esa tercera persona tendría acceso a la información intercambiada sin que el emisor se diera cuenta. Para resolver esto están las autoridades certificadoras (anexo P), las cuales permiten una conversación confiable y segura entre dos puntos.

Pero el problema de la privacidad no reside únicamente en el intercambio de la información de manera segura. También está en la regulación de esta información y el uso que se le da. Para ello existen leyes que están continuamente cambiando para asegurar la privacidad de los usuarios en internet, como veremos en la sección 2.6.

2.5.2 Anonimato

Para acceder a cualquier servicio de internet, el primer paso es identificarse, comúnmente a través de usuario y contraseña. Pero ¿por qué no puedo realizar una operación en internet de manera anónima? La respuesta es sencilla. En el caso de cometas alguna infracción o alguna ilegalidad sería imprescindible poder identificarte. Por esta razón el anonimato en internet es una utopía. Se usaría para aprovecharse y poder delinquir sin repercusiones en vez de usarse de la manera planteada. Es por eso por lo que entra en juego el concepto de anonimato justo, como veremos en la sección 2.5.6.

Para comenzar este apartado comenzamos viendo las diferencias entre el anonimato de información y de comunicación.

Anonimato de la comunicación
El anonimato de la comunicación consiste en que el emisor, o el receptor, o ambos tengan la capacidad de enviar mensajes de manera anónima. Esto no es posible a través de mensajes convencionales porque es necesario incluir la dirección IP de origen y de destino que impide el anonimato. Sin embargo, existen redes que pueden garantizar este anonimato en las comunicaciones.
Un ejemplo sencillo que ilustra esto es el problema de la cena de los tres criptógrafos el cual fue establecido por David Chaum en 1988. Otros dos ejemplos sobre este tipo de anonimato son las redes de mezcla y el enrutamiento de cebolla. A continuación, explicaremos brevemente estos tres ejemplos.

- **Cena de criptógrafos:**
  Consistía en que su cuenta había sido abonada de manera anónima. Entonces los tres criptógrafos querían saber si de verdad había sido pagado por uno de ellos sin que ninguno revelara la información de si ha pagado o no.

Para esto, cada par de criptógrafos establece un bit secreto (por ejemplo, lanzar una moneda). Luego cada criptógrafo calcula la función XOR con los bits secretos que han obtenido.
En el caso de que el criptógrafo haya pagado la cuenta, calcula la función NOR sobre resultado anterior. En estas condiciones, los tres criptógrafos comunican su resultado, y si tras realizar XOR con los tres resultados la solución es 0, significa que ninguno ha pagado, mientras que si es 1 significa que alguno de ellos lo hizo, aunque no ha perdido el anonimato.
El problema se puede generalizar al caso de n personas. En la Figura 2-2 vemos un ejemplo concreto en el que nadie paga, o paga un criptógrafo (A).

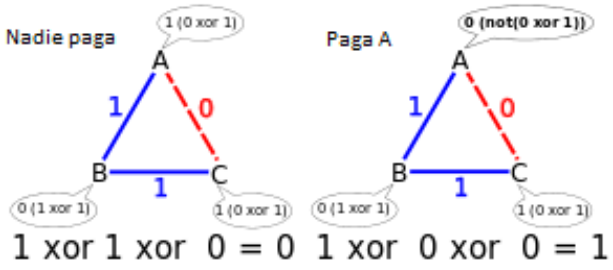

Figura 2-2: Cena de criptógrafos[1]

- **Redes de mezcla (Mix networks):**

  Son una familia de métodos de enrutamiento usados para conseguir el anonimato de los emisores. Estos envían su mensaje a unos servidores proxy llamados mixes que reciben los mensajes de varios usuarios, los mezclan y los envían al siguiente destino, que podría ser otro mix o la dirección del receptor. La Figura 2-3 es un ejemplo visual de una red de mezcla.

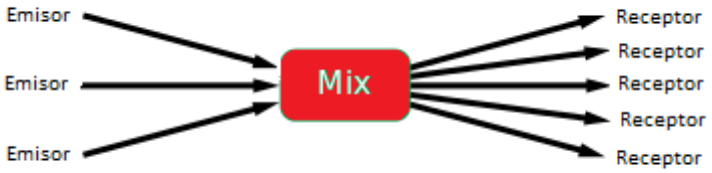

Figura 2-3: Red de mezcla[2]

- **Enrutamiento de cebolla:**

  El objetivo principal es separar la dirección de enrutado de los mensajes. Consiste en "esconder" la relación entre el origen y el destino a través de capas cifradas con criptografía pública, como podemos observar en la Figura 2-4.
  Para ello el mensaje sigue un camino de routers donde cada nodo lo que hace es descifrar con su clave privada ('pelar' la cebolla) y trasmitir los mensajes al siguiente nodo. Así se consigue que cada router solo tenga la información de cuál ha sido el

[1] Fuente: Imagen modificada. Original: Wikipedia. Autor: PauAmma.
[2] Fuente: Imagen modificada. Original: Wikipedia. Autor: Dani Shira.

nodo anterior de la ruta del mensaje y cuál será el siguiente. Esta es la estructura que sigue la red TOR (anexo Q).

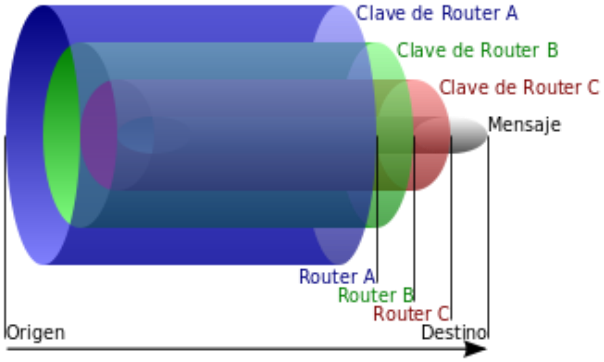

Figura 2-4: Enrutamiento de cebolla[3]

Anonimato de la información

El anonimato de la información consiste en no enviar la información relacionada con la identidad real de una persona.

En este TFG diseñaremos un protocolo de anonimia (sección 3.3) que garantizará el anonimato de la información. Para conseguir también el anonimato de comunicación, se plantea usar una red TOR para el envío de mensajes de dicho protocolo.

2.5.3 Conjunto de anonimia

Concepto usado para distinguir a los usuarios válidos o accesibles de un servicio. Aunque el servicio sea anónimo se debe restringir el uso a usuarios no válidos. Por ejemplo, en un foro anónimo sobre una clase de universidad, no tiene sentido que accedan personas ajenas a dicha clase, por lo que el conjunto de anonimia serían los alumnos y el profesor.

Para que un sistema garantice el anonimato, cada usuario válido debe de ser indistinguible a los demás usuarios del conjunto de anonimia. Esto implica que el conjunto de anonimato no tiene que ser muy pequeño para que los usuarios no sean identificados, aunque sea en pequeños grupos.

2.5.4 Pseudoanonimia

Consiste en usar pseudónimos como ID en vez de la identificación real. No supone una anonimia completa porque sigue haciéndolo distinguible dentro del conjunto de anonimia.

2.5.5 Trazabilidad

Capacidad que tiene el sistema de seguir las acciones de un determinado usuario, aunque no se identifique con su identidad real. Por ejemplo, si un usuario se identifica siempre con un pseudónimo, será muy sencillo trazarle por el sistema. Un buen sistema de anónima se requiere que sea no trazable.

[3] Fuente: Wikipedia. Autor: Harrison Neal HANtwister.

2.5.6 Anonimato justo

El anonimato justo es el estado al que se desea llegar en este TFG. Los usuarios se conectarán con el servicio de manera anónima, demostrando que realmente pertenecen al conjunto de anonimia, aunque no digan quiénes son realmente. Si un usuario comete una infracción, el sistema podrá revocar su identidad a partir de la prueba que acreditó al entrar, pero nunca lo podrá hacer si el usuario se comporta correctamente. Además, el revocar la identidad de un usuario no afectará a los demás miembros del conjunto de anonimia, que podrán seguir accediendo al sistema con total normalidad de manera anónima.

2.6 *Ley de Protección de Datos*

Para garantizar la privacidad en internet es necesario la implantación de leyes que regulen los datos de los clientes usados por las distintas aplicaciones. Estas leyes están en constante actualización debido a su importancia, como hemos podido comprobar en mayo de 2018 tras la implantación de la nueva ley europea de protección de datos [8].

Estas nuevas leyes sirven como motivación para destacar la importancia y actualidad del tema de la privacidad y anonimia en la red, el cual estamos tratando en este trabajo.

2.7 *Contratos inteligentes (Smart Contracts)*

Un contrato inteligente (del inglés: Smart Contract, SC) es un programa informático que ejecuta acuerdos entre dos o más partes de manera autónoma y automática.
Su objetivo principal es el de mejorar a los contratos tradicionales, aumentando su seguridad al actuar de manera objetiva por estar regulado por una máquina [21]. Además, suelen ser de código abierto para dar a conocer su proceso de funcionamiento interno.

Como ejemplo, podemos destacar Ethereum, que es una tecnología blockchain (ver sección 2.8) diseñada como una plataforma descentralizada de contratos inteligentes [22].

En este TFG se generarán contratos inteligentes para poder valorar las acciones de los clientes dentro del sistema. De esta forma, la anonimia de los clientes únicamente será revocada en el caso de que un contrato inteligente lo determine así.

2.8 *Cadena de bloques (Blockchain)*

La cadena de bloques (blockchain) es una estructura de datos moderna en la que la información se agrupa en bloques, los cuales forman una cadena al añadirles meta-información sobre otro bloque u otra cadena anterior en la línea temporal. Esto implica que, para modificar o borrar la información contenida en un bloque, es necesario editar los bloques posteriores en la línea temporal.
La estructura de datos blockchain actúa como una base de datos descentralizada y pública en la que los participantes, siguiendo un protocolo apropiado para realizar operaciones sobre dicha estructura, pueden alcanzar un consenso para garantizar la integridad de los datos almacenados sin necesidad de una entidad de confianza que centralice la información [24].

En este trabajo queremos tener en cuenta esta tecnología ya que es especialmente adecuada para almacenar de forma creciente datos ordenados en el tiempo, cuya confianza es distribuida y no depende de una entidad certificadora; como veremos con más detalle en la sección 5.4.

3 Análisis y diseño del sistema anónimo

En este capítulo analizaremos el diseño deseado del sistema. Comenzamos presentando los distintos actores que intervendrán. Después analizaremos los requisitos que se desea que el sistema siga. A partir de estos se explicará con detalle las fases en las que se divide el protocolo con el que se pretende conseguir la anonimia justa. Más adelante, se mostrará el diseño seguido en la aplicación, y la forma en la que se generarán y enviar los mensajes en el protocolo. Para finalizar se argumentará la generación de las claves con las que se podrá acceder al sistema.

3.1 *Actores y estructura del sistema*

El sistema se compone de 4 actores independientes. En la parte del cliente del servicio se encuentra el usuario (Ui), que será una aplicación Android. La parte del servidor se encuentra dividida en tres actores: un firmador (SU), un moderador (M), y el manager del servicio (SM) que realizan una funcionalidad diferente.
Los distintos actores se encargarán de gestionar la información de los clientes de manera independiente, y el principio básico que seguirá el sistema será conseguir el acceso anónimo a partir de esta división de información.

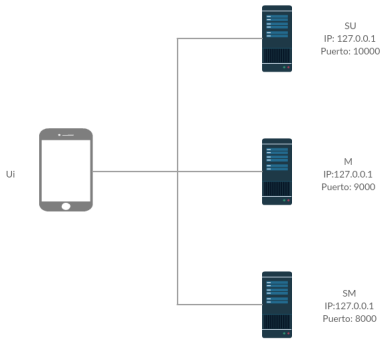

Figura 3-1: Estructura

- **Ui:** Es el cliente. Su objetivo es acceder al servicio de manera anónima.
- **SU:** Del inglés “Sign Up”. Su única funcionalidad es asignar a los usuarios válidos del sistema un identificador propio para realizar acciones dentro del sistema. Guardará la información real del usuario con su identificador para poder revocar su anonimato.
- **M:** Su única funcionalidad es evitar la trazabilidad (ver sección 2.5.5) del sistema, asignando a cada usuario que se identifique una clave de acceso única, según la forma en la que se explicará más adelante en la sección 3.7. Además, guardará la información del identificacor del usuario y esta clave de acceso.
- **SM:** Proporciona el servicio del sistema de manera anónima a los clientes que entreguen una clave de acceso al sistema válida. Se encarga también de comprobar las acciones del usuario dentro del sistema y en el caso de incumplir alguna norma del contrato, preguntará a M y SU acerca de la identidad real del usuario.

Una vez presentados los actores, introducimos las variables y nomenclatura que se usará en este capítulo:

- ID: Identificador númerico de 128 bits. Se usará la nomenclatura ID_0 para designar el primer ID obtenido por Ui.
- T: Marca temporal. Para distinguir entre las distintas marcas temporales usaremos la nomenclatura Ti donde i es un número natural.
- ValM: Es una cadena de caracteres aleatoria. También se denomirará token de validación. Sirve para controlar el acceso de los usuarios al moderador. El moderador será el único actor que se encargue de gestionar y repartir los ValM. Se usará la nomenclatura ValM_0 para designar el primer ValM obtenido por Ui.
- ValSM: Clave de acceso al sistema (o clave de acceso a SM). Tiene 256 bits de tamaño. M se encargará de crear nuevas claves de acceso mientras que SM se encargará de proporcionar el acceso a los Ui que entregen ValSM válidos.
- Smart Contract(SC): SU, M y SM serán SC (ver sección 2.7).
- Contrato: Indica una serie de condiciones que el Ui firma, comprometiéndose a cumplir mientras se encuentra dentro del sistema.
- Prueba: Indica la serie de condiciones que Ui ha incumplido en el Contrato.

3.2 *Requisitos del sistema*

Con el fin de conseguir los objetivos de la sección 1.2 se han definido los siguientes requisitos. Estos requisitos determinarán las cuatro distintas fases del protocolo de anonimia que procederemos a explicar en las siguientes secciones.

3.2.1 Requisitos funcionales

RF1- Aleatoriedad: Todos los actores podrán generar números o variables aleatorias seguras para realizar las distintas tareas del sistema.

RF2- Registro: SU podrá dar de alta a nuevos usuarios en el sistema. El listado de los nuevos clientes que podrán ser dados de alta estará incluido en una base de datos a la que sólo SU tendrá acceso. Para realizar el registro, los usuarios deberán firmar un Contrato con SU.

RF3- Generación de claves de acceso al sistema: Ui y M dispondrán de una función de código abierto determinista para generar las claves de acceso al sistema (ver sección 3.7).

> **RF3.1-** Además, M podrá generar y gestionar claves para que los usuarios puedan comunicase con él (ValM). Estas claves podrán ser solicitadas por SU y SM para dárselas a los clientes que las necesiten.

RF4- Anonimia: SM proporcionará los servicios del sistema a cualquier usuario que se identifique con una clave de acceso válida (ValSM).

RF5- Revocación: Gracias a la clave de acceso (ValSM), SM podrá preguntar a SU y M por la identidad real del usuario. Esta funcionalidad sólo se activará en caso de incuplimiento de Contrato.

> **RF5.1-** La clave de acceso al sistema corresponderá unívocamente con el acceso de un usuario en un momento determinado.

RF6- Separación de información: El acceso a las bases de datos se encontrará restringido.

> **RF6.1-** Los distintos actores no podrán compartir cualquier información relacionada con los usuarios del sistema a excepción del RF5.

RF7- Notificación al usuario: Se notificará al usuario en caso de que su anonimato sea revocado.

3.2.2 Requisitos no funcionales

RNF1- Seguridad: Todas las comunicaciones entre los distintos actores del sistema irán encriptadas por RSA o AES.
RNF2- Seguridad: El emisor utilizará una firma RSA para garantizar la autenticidad de los mensajes.
RNF3- Seguridad: Se incluyen marcas temporales para evitar ataques de réplica.
RNF4- Seguridad: Los datos guardados por los distintos actores deberán de estar protegidos evitando así el robo de información por parte de terceros.
RNF5- Seguridad: Las claves de acceso al sistema tendrán una fecha de caducidad y deberán renovarse cada cierto tiempo. Además, tendrán un tamaño suficientemente grande para evitar colisiones.
RNF6- Seguridad: Las claves de acceso al sistema se generarán con el fin de no poder rastrear a los usuarios a partir de ellas. Eso implica que pocos cambios en las entradas de la función generadora producirán importantes cambios en la salida.
RNF7- Usabilidad: La interfaz usada por los usuarios del servicio será sencilla e intuitiva.
RNF8- Usabilidad: Las distintas acciones por parte del sistema para conseguir el acceso anónimo serán transparentes para el usuario.
RNF9- Tiempo respuesta: El acceso correcto al sistema deberá de ser rápido y no podrá demorarse más de 2 segundos.
RNF10- Acceso: El sistema debe permitir múltiples conexiones simultáneas sin saturarse.

3.3 *Diseño del protocolo de anonimia*

El protocolo de anonimia que seguirá el sistema se divide en cuatro fases distintas, las cuales procedemos a explicar.
La estructura de los mensajes enviados durante todas las fases del protocolo seguirá el diseño de la sección 3.5 y el envío de mensajes seguirá la nomenclatura de la sección 3.6.

3.3.1 Fase 1 - SETUP

Es la primera fase del protocolo de anonimia (ver Figura 3-2). En esta fase está basada en el **RF2**, es decir, sirve para registrar a un usuario en el sistema. Sentará las bases y distribuirá las variables y tokens necesarios para poder consumir el servicio que ofrecerá SM.

Prerrequisitos: Ninguno.
Actores que intervienen: Ui, SU y M.
Requisitos funcionales: RF1, RF2, RF6.

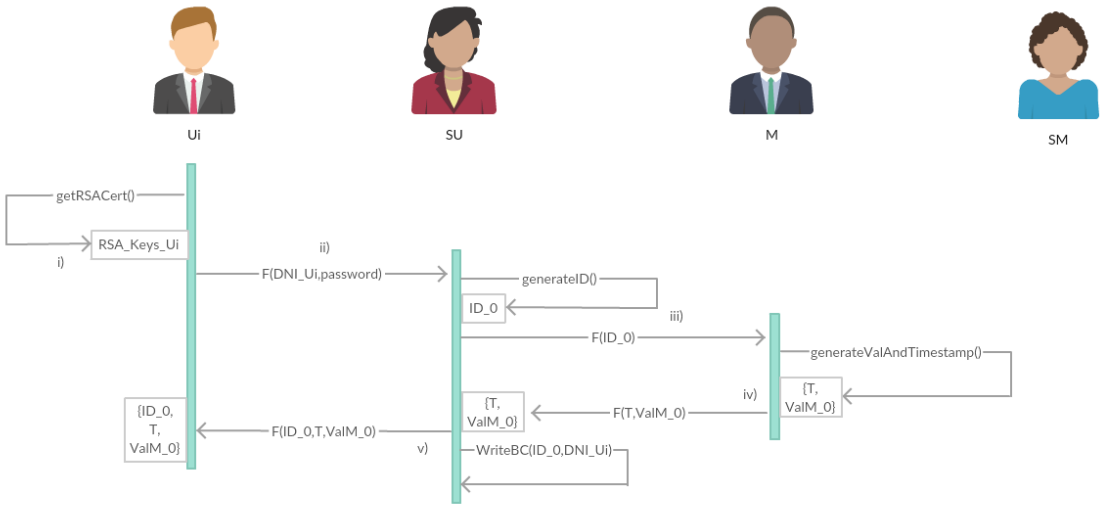

Figura 3-2: Fase 1

i. El Usuario (Ui) genera su par de claves RSA asociadas a su identidad real.
ii. Ui envía a SU el Contrato firmado del servicio, en donde se incluye la información personal de Ui como podría ser su DNI y una contraseña de acceso al sistema. Este Contrato son una serie de condiciones que el cliente se compromete a cumplir dentro del servicio.
iii. SU comprueba que el usuario se encuentra en la lista de usuarios válidos del sistema y que este no ha sido registrado previamente. En el caso correcto, SU generará un ID_0 que será un número de 128 bits único y le enviará dicho identificador a M.
iv. M genera una marca temporal (T) y la guardará junto al ID_0 que acaba de recibir en una base de datos de clientes. También generará un token de validación (ValM_0) con el cual permitirá a los usuarios acceder a M y lo guardará en una base de datos de tokens. Después enviará a SU estas dos variables que acaba de calcular.
v. SU entregará al cliente las variables ID_0, T y ValM_0, completando así su registro. Estas variables serán los elementos suficientes y necesarios que el usuario necesitará en las siguientes fases del protocolo. Para finalizar, guardará la información real del cliente junto a su ID_0 en una base de datos, pero se deja diseñado para utilizar la tecnología blockchain (ver sección 2.8) en su lugar. La motivación sobre el uso de blockchain vendrá explicada en la sección 5.4.

Resultados de esta fase:
El acceso en esta fase no ha sido anónimo, ya que el usuario ha enviado a SU su identidad real.
Se han generado tres variables (ID_0, T y ValM_0) que, a partir de este momento, será la única información personal que Ui aporte para acceder al servicio.

Ganancia de información sobre Ui de cada actor en esta fase:
SU ha conseguido información sobre la identidad real del usuario, su identificador, su marca temporal y su primer token de acceso a M.
M ha conseguido información sobre el ID del usuario y su marca temporal T pero desconoce su identidad real. Tambien conoce el token con el que podrá hablar con él en la siguiente fase.

3.3.2 Fase 2 – Acceso anónimo

Es la segunda fase del protocolo de anonimia (ver Figura 3-3). Está basada en la idea de evitar la trazabilidad de acceso al servicio ofertado por SM por parte del usuario. En esta fase se implementa el **RF3** con el que Ui consigue una clave de acceso única. Requiere ser ejecutado siempre como paso previo al acceso al sistema.

Prerrequisitos: El usuario se ha registrado en el sistema mediante la fase 1.
Actores que intervienen: Ui y M.
Requisitos funcionales: RF1, RF3, RF6.

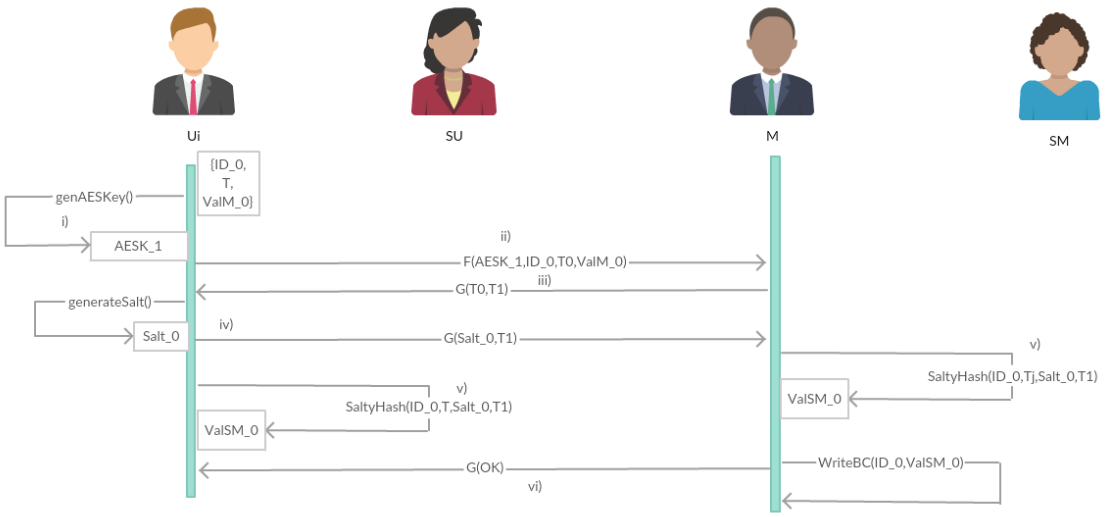

Figura 3-3: Fase 2

i. Ui genera una clave AES y la guarda bajo la variable AESK_1.
ii. Se produce un intercambio de información con M en tres pasos. En el primero, cifra con la clave pública de M la clave simétrica AESK_1, una marca temporal T0, el identificador del usuario ID_0 y el token de validación ValM_0. Este mensaje también estará firmado por Ui para evitar ataques de réplica del mensaje.
iii. M devuelve a Ui un mensaje con la marca temporal recibida (T0), y una nueva que genera justo en este momento (T1). A partir de este paso, los mensajes irán cifrados con la clave simétrica AESK_1.
iv. Ui generará un salt (Salt_0), el cual se usará para aumentar la entropía en la función generadora de claves del sistema de la sección 3.7, y se lo enviará a M junto a la marca temporal T1.
v. Tanto M como Ui calcularán en paralelo un token de validación (ValSM_0) usando los valores ID_0, T1, Salt_0 y T (ver sección 3.7) (Aclaración sobre la obtención de T1 por parte de M: Recordemos que en el paso iv de la fase 1 del protocolo, M había guardado el ID_0 y T1 en una base de datos de clientes).
vi. Si todo va bien, M le envía un mensaje a Ui indicando la finalización del protocolo, y guarda la clave de acceso calculada junto al identificador del cliente en en una base de datos, pero se deja diseñado para utilizar la tecnología blockchain (ver sección 2.8) en su lugar. Además, escribirá ValSM_0 en una base de datos a la que tendrá acceso SM.

Resultados de esta fase:
El acceso no ha sido anónimo. Ui ha utilizado un ID_0 para acceder a M, luego se pueden trazar sus acciones dentro de esta fase, pero ya se observa un cambio con respecto a la primera fase.
Se ha generado una clave ValSM_0 de un único acceso para que Ui pueda consumir el servicio. A partir de esta clave no se puede sacar ninguna información acerca de Ui.

Ganancia de información sobre Ui de cada actor en esta fase:
M ha conseguido información sobre el ValSM asociado al ID de un usuario del sistema.

3.3.3 Fase 3 – Acceso al servicio

Este protocolo realiza el acceso al servicio de manera anónima por parte del usuario (ver Figura 3-4). La forma en la que el usuario se identificará será únicamente mediante el token ValSM_0. Después de que el usuario indique una petición al servidor de desconexión, se comprobará la validez de las acciones dentro del sistema. Por un lado, si son correctas se le devolverá un nuevo token de acceso al moderador, ValM_1 para que pueda solicitar de nuevo el acceso anónimo al servicio. Por otro lado, si no lo son se pasará a ejecutar el protocolo 4.

Prerrequisitos: El usuario ha realizado el acceso anónimo mediante el protocolo 2.
Actores que intervienen: Ui, SM y M.
Requisitos funcionales: RF1, RF4, RF6.

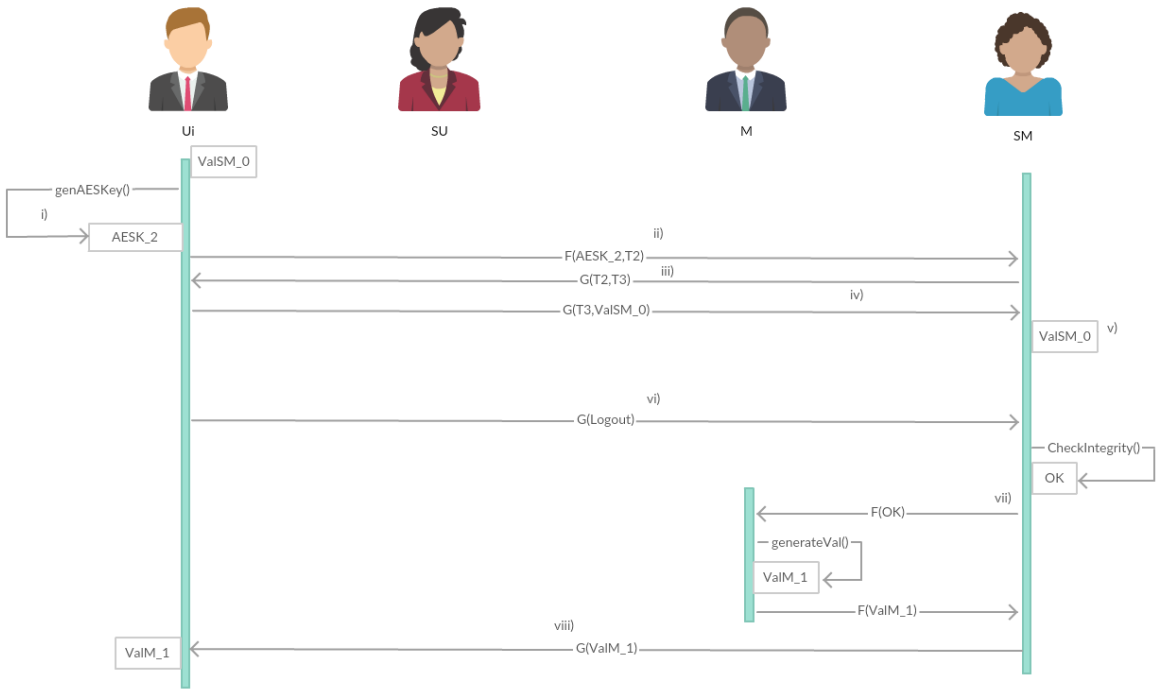

Figura 3-4: Fase 3

i. Ui genera una clave AES y la guarda bajo la variable AESK_2.
ii. Se produce un intercambio de información con SM en tres pasos. En el primero, cifra con la clave pública de SM la clave simétrica AESK_2 y una marca temporal T2. Este mensaje también estará firmado por Ui para evitar ataques de réplica del mensaje.
iii. SM devuelve a Ui un mensaje con la marca temporal recibida (T2), y una nueva que genera justo en este momento (T3). A partir de este paso, los mensajes irán cifrados con la clave simétrica AESK_2.
iv. Ui enviará a SM la marca temporal recibida (T3) y el token de validación ValSM_0.
v. SM comprueba que el token de validación es correcto, y lo elimina de la base de datos. A partir de este punto, el usuario habrá accedido correctamente al sistema, y podrá realizar las acciones que considere oportunas.
vi. Una vez el usuario acabe de hacer sus operaciones, envía un mensaje de desconexión a SM.
vii. SM comprueba que el usuario ha cumplido con el contrato que firmó en el protocolo 1 mediante la función CheckIntegrity. Si la respuesta de esa función es OK, enviará un mensaje a M cifrado y firmado con RSA solicitando un nuevo token de acceso (ValM_1).
viii. SM le enviará ValM_1 a Ui y dará por finalizado el protocolo de acceso al servicio.

Resultados de esta fase:
Ui se ha conectado con SM de manera completamente anónima. SM no puede sacar ningún tipo de información sobre el usuario, solo sabe que posee una clave válida para poder consumir el servicio que ofrece.
Acaba la fase otorgando a Ui un nuevo ValM_1 para que pueda repetir la fase 2 del protocolo cuando desee acceder de nuevo al sistema.

Ganancia de información sobre Ui de cada actor en esta fase:
SM sabe que Ui ha aportado una clave de acceso válida, pero no sabe quién es.
M sabe que un cliente ha recibido un nuevo ValM para poder acceder a él, pero no sabe quién.

3.3.4 Fase 4 – Revocación de identidad

Este protocolo se ejecuta en el caso de que la función CheckIntegrity devuelva una prueba de una acción ilegal por parte del usuario (ver Figura 3-5). SM preguntará a M y SU acerca de la identidad real de Ui, y le notificará que su identidad ha sido revocada. El usuario no podrá acceder de nuevo al servicio porque ya no dispondrá de ningún token de acceso.

Prerrequisitos: El usuario tiene que haber accedido al servicio (es decir, se encuentra en el paso vi de la fase 3 del protocolo) y la función CheckIntegrity devuelve una Prueba de las acciones ilegales de Ui en este acceso.
Actores que intervienen: Ui, SM, M y SU.
Requisitos funcionales: RF1, RF5, RF5.1, RF7.

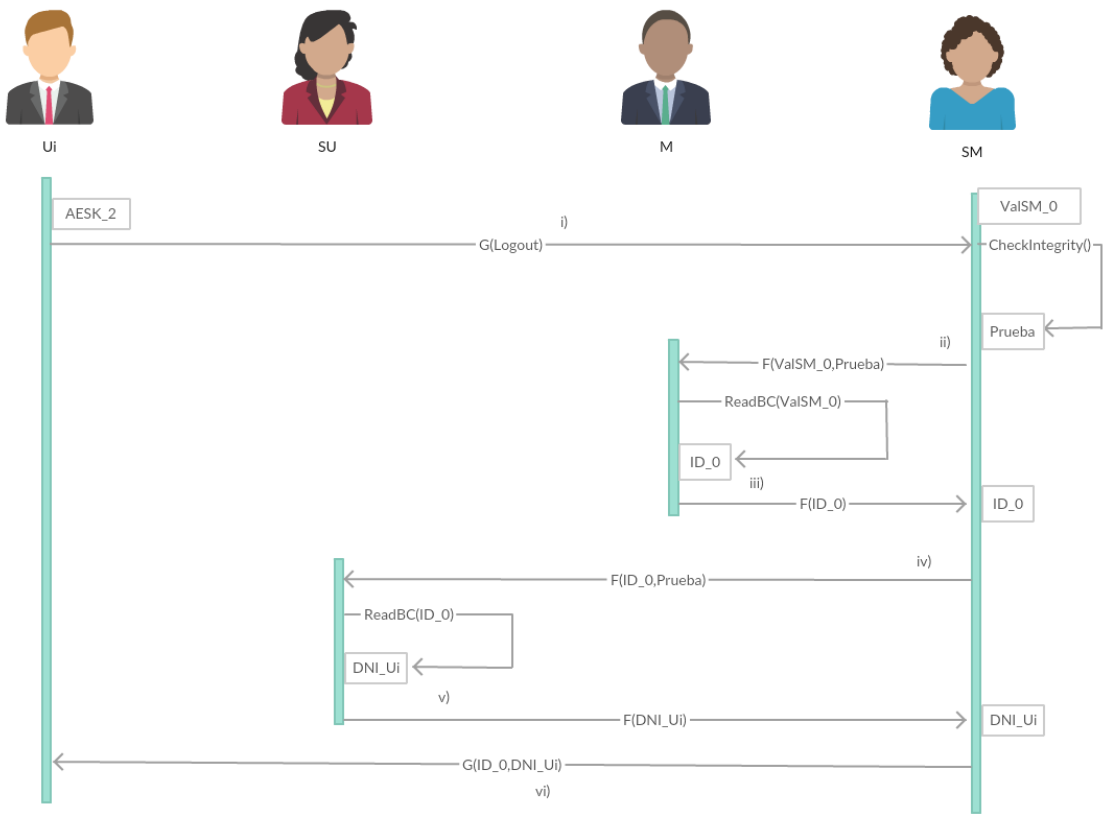

Figura 3-5: Fase 4

i. El Usuario envía un mensaje de desconexión a SM, cifrado con clave simétrica.
ii. SM comprueba que el usuario ha cumplido con el contrato que firmó en la fase 1 (ver sección 3.3.1) mediante la función CheckIntegrity. Si Ui ha infringido el contrato durante su sesión, la función CheckIntegrity devolverá una Prueba, la cual se enviará a M en un mensaje cifrado y firmado mediante RSA junto con la clave ValSM_0.
iii. M leerá en su base de datos el ID_0 que se encuentra asociado a esa clave, y se lo envía a SM.
iv. SM envía a SU la Prueba y el ID_0 cifrado y firmado con RSA.
v. SU leerá en su base de datos los datos reales del usuario (DNI_Ui) que se encuentran asociados a ese identificador, y se lo devuelve a SM.
vi. SM le envía un mensaje a Ui cifrado con la clave simétrica AESK_2 indicándole que su identidad ha sido revocada.

Resultados de esta fase:
SM se ha dado cuenta de que Ui ha incumplido su contrato. Tras pedir la información a M y SU consigue revocar su anonimia y descubrir la identidad real del usuario.
Al final de esta fase, notifica al usuario de esto, y no le devuelve un nuevo token ValM_1 como en la fase 3. Esto hace que Ui no pueda volver a solicitar una clave de acceso a M porque no tiene un token para acceder a él, luego se le impediría el futuro acceso al servicio.

Ganancia de información sobre Ui de cada actor en esta fase:
SM logra revocar la anonimia de Ui y tener su información real.
M sabe que un ID del sistema ha incumplido el contrato.
SU sabe la identidad real y el ID de un usuario que ha incumplido el contrato.

3.4 *Aplicación Android*

Como servicio de prueba del protocolo explicado anteriormente, se ha diseñado una aplicación Android que ofrezca el servicio de la creación y visualización de temas y mensajes en un foro anónimo.

En la Figura 3-6 mostramos el esquema de actividades que seguirá muestra aplicación.

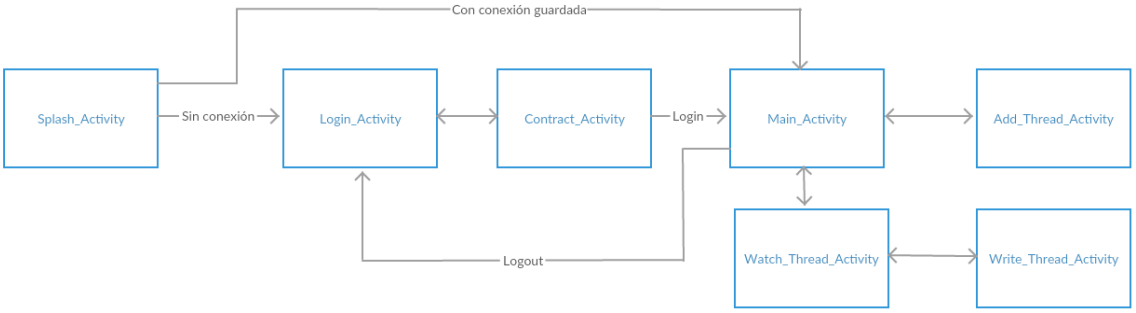

Figura 3-6: Esquema de actividades

- **Splash_Activity:** Actividad que mostrará una imagen de presentación de la aplicación mientras se carga. Si no hay conexión guardada se cargará Login_Activity. En el caso contrario, se pasará directamente a Main_Activity.
- **Login_Activity:** Es la actividad que permite a los usuarios loguearse en el sistema. Requerirá la firma del contrato de Contract_Activity para realizar el login, y pasará a Main_Activity tras ejecutar la Fase 1 del protocolo.
- **Contract_Activity:** Actividad que ofrece al usuario la capacidad de poder ver y firmar el contrato del sistema.
- **Main_Activity:** Actividad principal del sistema. Ejecutará las Fases 2 y 3 del protocolo para acceder al sistema de manera anónima y mostrará al usuario un listado de hilos abiertos del foro. En esta actividad, el usuario podrá leer los mensajes de un hilo, añadir un nuevo hilo o desconectarse (donde se ejecutará la Fase 4 del protocolo).
- **Add_Thread_Activity:** Actividad que aporta la funcionalidad para añadir un hilo al foro.
- **Watch_Thread_Activity:** Actividad que aporta la funcionalidad de leer un hilo existente.
- **Write_Thread_Activity:** Actividad que aporta la funcionalidad de escribir un mensaje en un hilo.

Se incluye en este trabajo el anexo (anexo R) en el que añadimos información adicional sobre los métodos e interfaces usadas en cada actividad.

3.5 *Estructura mensajes protocolo*

Todos los mensajes del protocolo seguirán una misma estructura (como se muestra en la Figura 3-7) para facilitar la escritura y lectura de estos mensajes. Comenzarán con un identificador numérico que determinará el mensaje del protocolo al que corresponde. Le seguirá un carácter especial ';' para distinguir este identificador con el cuerpo del mensaje. En el cuerpo del mensaje irán todos los datos que se transmitan en forma de cadena de caracteres y se utilizará otro carácter especial ':' para separar estos datos.

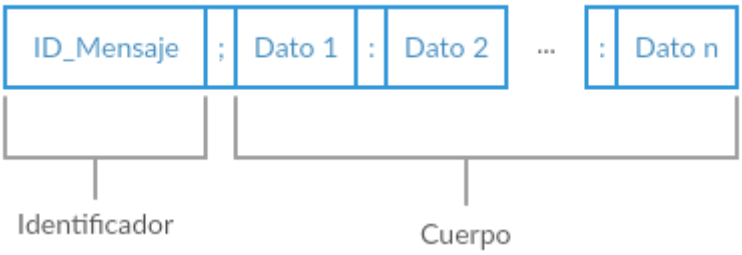

Figura 3-7: Estructura mensajes

Pero estos mensajes necesitan ser cifrados durante el protocolo. Sea S la nomenclatura usada para definir un mensaje que se quiera enviar. Definimos las funciones de envío de mensajes en la siguiente sección.

3.6 *Diseño envío mensajes protocolo*

Debido al requisito no funcional **RNF1** (ver sección 3.2.2), todos los mensajes del protocolo irán cifrados. Podemos definir dos esquemas de cifrado, los cuales han sido referenciados en la sección 3.3 para evitar la repetición de información.

3.6.1 Cifrado y firma RSA

Este esquema de cifrado se usará en todos los mensajes cifrados con clave asimétrica. El esquema es el de la Figura 2-1.
$Sea\ PrK_A := Clave\ Privada\ del\ actor\ A.$
$Sea\ PuK_A := Clave\ Pública\ del\ actor\ A.$
$Sea\ HASH(S) := Función\ hash\ con\ entrada\ el\ mensaje\ S.$
$Sea\ E_{KEY}(S) := Encriptación\ RSA\ del\ mensaje\ (S)\ con\ clave\ KEY.$
$Definimos\ F_{A\to B}(S) :=\ E_{PrK_A}\big(HASH(S)\big)||E_{PuK_B}(S)$ donde | | denota concatenación.

Entonces $F_{A\to B}(S)$ es la función usada para el envío de mensajes cifrados con RSA con origen en el actor A y destino el actor B.

3.6.2 Cifrado AES

Este esquema de cifrado se usará en todos los mensajes cifrados con clave simétrica.
$Sea\ E_{AEK}(S) := Encriptación\ AES\ del\ mensaje\ (S)\ con\ clave\ \ simétrica\ AEK.$
$Definimos\ G_{A\to B}(S) :=\ E_{AEK}(S).$

Entonces $G_{A\to B}(S)$ es la función usada para el envío de mensajes cifrados con AES con origen en el actor A y destino el actor B.

3.7 *Creación de la clave de acceso al sistema (ValSM_0)*

La creación de la clave de acceso al sistema (ValSM_0) es uno de los puntos más críticos de todo el protocolo. La idea es encontrar una función *H(S)=K*, donde *S* son un número determinado de entradas y K la clave de acceso al sistema que cumpla los siguientes requisitos ya explicados en la sección 3.2:

- **RF1:** La clave se tiene que generar de forma aleatoria segura. Es decir, debe de ser un conjunto de bits originados de la forma más impredecible posible, evitando así que una tercera persona ataque al sistema a partir del conocimiento de la secuencia aleatoria producida por una función generadora de variables pseudo-aleatorias.
- **RF3:** La función que genera esta clave de acceso tiene que encontrarse en Ui y M. Además, esta función debe ser determinista, para que produzca la misma clave a partir de los mismos valores de entrada.
- **RF5.1:** La clave generada tiene que es única para un determinado acceso de un determinado cliente.
- **RNF5:** La función debe generar una clave lo suficientemente grande para conseguir con cierta probabilidad matemática evitar las colisiones de claves.
- **RNF6:** La salida de la función no debe dar ninguna información sobre las entradas. También se debe producir importantes cambios en la salida a partir de pocos cambios en las entradas para evitar el rastreo de las entradas de la función a partir de salidas similares.
- **RNF9:** Esta función no puede ser muy costosa computacionalmente, ya que el tiempo de respuesta debe de ser pequeño.

Empezamos analizando las entradas de la función. Para cumplir **RF5.1** las entradas serán ID_0, T1, Salt_0 y T de acuerdo con lo explicado en la fase 2 del protocolo de la sección 3.3.2. De esta forma, las entradas serán distintas de acuerdo con el usuario y el momento de acceso.
Mirando ahora la salida, es decir la clave, teniendo en cuenta el requisito **RNF5** dicha clave tendrá un tamaño de 256 bits por diseño.
Entonces sólo falta encontrar la función H que cumpla el resto de requisitos propuestos. La solución propuesta en este TFG ha sido una función hash (SHA-256) cuya entrada sea la cadena de caracteres formada por la concatenación de todas las entradas.

Esta solución es capaz de resolver todos los requisitos gracias a las propiedades de las funciones hash criptográficas explicadas en la sección 2.2.
Se cumple el **RF1** ya que las funciones hash pueden ser usadas para generar valores aleatorios seguros. **RF3** se cumple ya que las funciones hash son deterministas, y además se aprovecha el hecho de que esa función hash ya estaba siendo usada en Ui y M para realizar firmas RSA. **RNF6** se cumple ya que las funciones hash son denominadas funciones de una sola vía y además poseen la propiedad de efecto cascada. También se cumple **RNF9** ya que las funciones hash tienen poco coste computacional, como hemos podido observar en la Tabla 2-1.

4 Desarrollo e integración

En este trabajo se han usado las siguientes tecnologías de la información: C (openssl), Python, Java, criptosistemas de clave simétrica y asimétrica, funciones hash criptográficas, firmas digitales, PostgreSQL, Smart Contracts.

Comenzamos detallando la funcionalidad desarrollada para cada uno de los actores, además de los esquemas de las bases de datos PostgreSQL usadas.

Ui

Este actor es que ejecuta toda la parte del cliente del servicio. Es por ello por lo que tiene incluida toda la funcionalidad desarrollada en Java. Como podemos ver en la Figura 4-1 en él se encuentran métodos usados para encriptar y desencriptar siguiendo el algoritmo RSA. También se incluye un método para calcular la salida de la función hash SHA-256 a una cierta entrada. Además, se incluyen métodos para generar una firma RSA y para poder verificarla también.
En la parte relacionada al cifrado simétrico, se encuentran métodos usados para encriptar y desencriptar mensajes siguiendo el algoritmo AES. Para finalizar, este actor también posee un método para generar claves AES.
En cuanto a la base de datos de este cliente, únicamente se tiene que preocupar de sus variables usadas para cada sesión, por lo que no necesitará el uso de bases de datos para guardar información.

Figura 4-1: Esquema Ui

SU

Este actor tiene la funcionalidad de encriptar y desencriptar mensajes en RSA, además de poder generar también firmas RSA y poder verificarlas. También podrá generar números aleatorios de 128 bits que serán usados como ID.
En cuanto al módulo de AES, este actor no necesitará ninguna funcionalidad de acuerdo con el diseño del protocolo de las secciones 3.3.1 y 3.3.4.
SU tendrá acceso de lectura a una base de datos que contiene la información real de los usuarios que pueden acceder al sistema, incluyendo una contraseña de acceso que, por hipótesis, suponemos que ha sido entregada previamente a los clientes. También tendrá el acceso a un blockchain permisivo en el que guardará las asignaciones de los ID generados con el DNI de cada usuario. La Figura 4-2 muestra toda la funcionalidad de este actor.

Figura 4-2: Esquema SU

M

Como se muestra en la Figura 4-3, el moderador contendrá los métodos necesarios para encriptar y desencriptar mensajes tanto en RSA como en AES. También dispondrá de métodos usados para la creación y verificación de firmas digitales. De acuerdo con el requisito **RF3** y la sección 3.7, este actor también tendrá un método para calcular el resultado de una función hash, previamente acordada.

En la parte de bases de datos, este actor tendrá acceso de lectura y escritura a bases de datos que guardan la información de los clientes registrados en el sistema, y los tokens de acceso con él. También tendrá el permiso de escritura a la base de datos usada por SM llamada Token_SM en la Figura 4-4. Por último, tendrá el acceso a un blockchain permisivo en el que guardará las claves de acceso generadas junto al ID del usuario implicado.

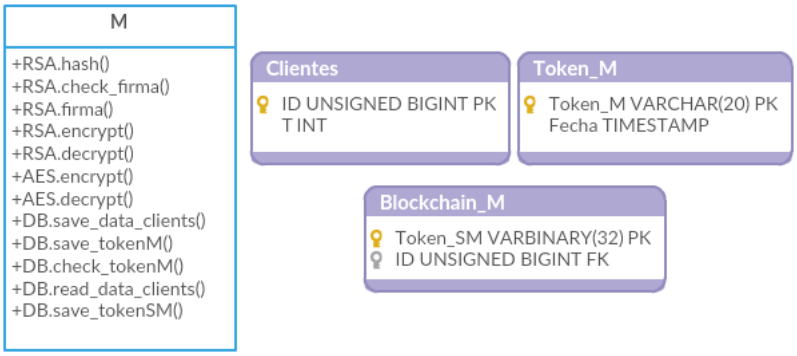

Figura 4-3: Esquema M

SM

Este último actor tendrá los métodos necesarios para realizar todas las encriptaciones en clave simétrica y asimétrica ya detallados anteriormente, como se puede observar en la Figura 4-4. Además, tendrá acceso a la base de datos de tokens de SM para poder ofrecer el servicio a los usuarios correctos.

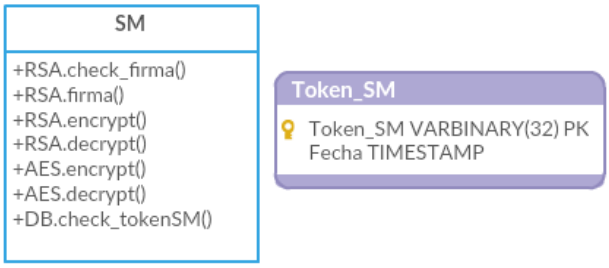

Figura 4-4: Esquema SM

4.1 *Generación de claves RSA*

La generación de un par de claves RSA se puede hacer de varias formas y estas pueden ser guardadas en varios formatos. En este TFG se han generado claves en dos diferentes formatos, siendo usados en los dos diferentes lenguajes de programación.
Por una parte, para la implementación en Python, las claves se han generado y guardado en formato PEM. Este formato se caracteriza por que el fichero de claves puede ser leído cómodamente yo el usuario, ya que las claves se guardan con caracteres alfanuméricos (está cifrado en Base 64) y se encuentran entre las cadenas de caracteres "-----BEGIN RSA PRIVATE KEY-----" y "-----END RSA PRIVATE KEY-----".
Por otro lado, en Java, se ha utilizado el formato PKCS8. Esto ha servido para aprender el manejo de las claves en diferentes formatos. Este formato es más invisible para el usuario, ya que el fichero de claves no se puede leer correctamente porque se encuentra escrito en modo binario.

4.1.1 OpenSSL

OpenSSL es un conjunto de herramientas a nivel comercial con toda la funcionalidad de la seguridad en la capa transporte (STL) y la capa de puertos seguros (SSL) [5].
OpenSSL permite la creación de certificados, además de la exportación de claves RSA en distintos formatos. Ahora, mostramos unos ejemplos de uso siguiendo el manual de [6]:

Generación de un par de claves RSA

```
$ openssl genrsa -out RSAkeys.key 8192
```

Este comando genera un par de claves RSA de tamaño 8192 bits y las guarda en el archivo "RSAkeys.key". Estas claves se encuentran en formato PEM.

Exportación de la clave pública

```
$ openssl rsa -in RSAkeys.key -pubout > RSApublica.pub
```

Este comando extrae la clave pública del par de claves almacenadas en "RSAkeys.key" y la deposita en el archivo "RSApublica.pub". Seguirá manteniendo el formato PEM.

Creación de un certificado

```
$ openssl req -new -key RSAkeys.key -out RSAkeys.csr
```

Este comando genera una petición de firma de certificado para que una autoridad certificadora pueda firmar que mi clave pública es auténtica y lo guarda en el fichero "RSAkeys.csr".

Cambio de formato de claves

```
$ openssl pkcs8 -inform DER -nocrypt -in RSAkeys.p8 -out RSAkeys.pem
```

Este comando consigue cambiar el formato de un par de claves RSA de PKCS8 (RSAkeys.p8) a PEM (RSAkeys.pem).

4.1.2 Java

Para generar un par de claves RSA en java utilizamos el paquete *java.security*, el cual contiene las clases *KeyPairGenerator* (KPG) y *KeyPair* (KP). A partir del manual de uso de la librería de seguridad de java [7], podemos indicar los pasos a seguir.

Crear el KPG:

```
KeyPairGenerator keyGen = KeyPairGenerator.getInstance("RSA");
```

Inicializar el KPG:

Se puede inicializar el KPG, aunque en la mayoría de casos se usa la inicialización específica del algoritmo. En este TFG optaremos por la segunda opción.

Generar la Pareja de Claves(KP):

```
KeyPair pair = keyGen.generateKeyPair();
```

4.1.3 Python

En Python, un par de claves RSA se puede generar usando el módulo *RSA* de la librería *Crypto.PublicKey*. Esta librería contiene una función llamada *generate*, cuyos detalles son los siguientes:

generate(bits, randfunc=None, progress_func=None, e=65537)

Genera un nuevo objeto RSA key.

Parámetros de entrada:

- bits(int): longitud de la clave
- randfunc(callable): función generadora de números aleatorios.
- progress_func(callable): función opcional que será llamada con una pequeña cadena de caracteres que contenga el parámetro de clave que actualmente se está generando.
- e(int): exponente público del algoritmo RSA.

Salida:

- Un RSA key object (_RSAobj). Esa clase se encuentra también en el módulo RSA previamente mencionado.

Para generar un par de claves RSA lo único que se tiene que hacer es llamar a esta función. Después, si se quieren guardar estas claves, se puede llamar a la función *exportKey* de *_RSAobj*, eligiendo el formato deseado. En el caso particular de este trabajo, se guardarán en formato PEM.

4.2 *Algoritmos de encriptación*

Los algoritmos de encriptación usados en este trabajo han sido RSA y AES. No obstante, se ha necesitado utilizar una librería con el mismo algoritmo para realizar esta encriptación para poner en común tamaño de claves, entrada y salida.

4.2.1 Java

El paquete usado en Java para realizar la encriptación de mensajes deseada se encuentra en la clase *Cipher* del paquete *javax.crypto*. Gracias a la información en [7] podemos indicar que los pasos a seguir son los siguientes:

RSA

Se obtiene la clase para encriptar/desencriptar:

Para ello, obtenemos una instancia, que en este TFG será el algoritmo que realiza el cifrado RSA en modo ECB como mostramos en la siguiente línea de código.

Cipher rsa=Cipher.getInstance("RSA/ECB/PKCS1Padding");

Encriptación:

Para encriptar un mensaje (texto_claro), primero tendemos que generar una clave pública (publicKey) y podemos obtener un array de bytes como resultado con el siguiente código, utilizando el modo ENCRYPT_MODE del cifrado.

rsa.init(Cipher.ENCRYPT_MODE, publicKey);
byte[] texto_encriptado = rsa.doFinal(texto_claro);

Desencriptación:

El proceso de desencriptar un mensaje es similar al de encriptar, lo que cambia es el modo de cifrado (DECRYPT_MODE), y el uso de la clave privada (privateKey) previamente generada.

rsa.init(Cipher.DECRYPT_MODE, privateKey);
byte[] texto_claro = rsa.doFinal(texto_encriptado);

AES

Inicialización de variables:

El cifrado AES se realiza de forma similar a RSA, aunque previamente hay que inicializar dos variables que se usarán en el cifrado. Para el vector de inicialización se usará la clase *IvParameterSpec*, mientras que para la clave simétrica usaremos la clase *SecretKeySpec*, ambas clases incluidas en el paquete *javax.crypto.spec*. Para realizar esta fase, suponemos que tenemos dos arrays de bytes con la información del vector de inicialización (initVector) y de la clave (key). Entonces el código es el siguiente:

IvParameterSpec iv = new IvParameterSpec(initVector);
SecretKeySpec keySpec = new SecretKeySpec(key);

Se obtiene la clase para encriptar/desencriptar:

Para ello, obtenemos una instancia, que en este TFG será el algoritmo que realiza el cifrado AES en modo CBC como mostramos en la siguiente línea de código.

Cipher aes=Cipher.getInstance("RSA/CBC/PKCS5PADDING");

Encriptación:

De manera similar al cifrado anterior, para cifrar un mensaje (texto_claro) necesitamos usar el modo ENCRYPT_MODE del cifrador y pasarle las variables keySpec e iv, previamente inicializadas.

aes.init(Cipher.ENCRYPT_MODE, keySpec,iv);
byte[] texto_encriptado = aes.doFinal(texto_claro);

Desencriptación:

Para recuperar el mensaje original, el proceso es el mismo, únicamente cambiando al modo DECRYPT_MODE.

aes.init(Cipher.DECRYPT_MODE, keySpec,iv);
byte[] texto_claro = aes.doFinal(texto_encriptado);

4.2.2 Python

RSA

Para realizar el cifrado RSA se ha usado la clase *PKCS1_v1_5* del paquete *Crypto.Cipher*, la cual realiza la misma codificación que el algoritmo de cifrado implementado en Java, consiguiendo así que el proceso de encriptar y desencriptar mensajes entre los distintos actores se haga correctamente, independientemente del lenguaje de programación utilizado para implementarlos. Los pasos seguidos para realizar este cifrado son los siguientes:

Obtención de clave:

Como vimos en la sección 4.1, el formato de claves usadas en Python es PEM. Entonces, utilizando la clase *RSA* del paquete *Cripto.PublicKey* podemos conseguir la clave a partir de la variable key, la cual suponemos que contiene la información de la clave.

keyRSA=RSA.importKey(key,'PEM')

Se obtiene la clase para encriptar/desencriptar:

Únicamente hay que inicializar la clase *PKCS1_v1_5*, utilizado como entrada la clave RSA.

cipher=PKCS1_v1_5.new(keyRSA)

Encriptación:

Para encriptar un mensaje (mensaje_claro) solo hay que utilizar el método *encrypt*.

mensaje_cifrado=cipher.encrypt(str.encode(mensaje_claro))

Desencriptación:

El proceso de desencriptar es similar, utilizando ahora el método *decrypt* sobre el array de bytes que compone el mensaje cifrado (mensaje_cifrado)

mensaje_claro=cipher.decrypt(mensaje_cifrado)

4.3 *Funciones hash*

La función hash implementada ha sido SHA-256. Esta función se usará como la función resumen de la firma RSA de la Figura 2-1 y como función para generar las claves de acceso al sistema siguiendo el diseño de la sección 3.7.

4.3.1 Java

Las funciones hash en Java se encuentran en la clase *MessageDigest* del paquete *java.security*. Para realizar la función resumen SHA-256 de un array de bytes (datos), hay que obtener la instancia de la función hash y obtener su digest de la forma que mostramos a continuación:

```
MessageDigest resumen = MessageDigest.getInstance("SHA-256");
byte[] salida = resumen.digest(datos);
```

4.3.2 Python

En la clase *SHA256* del paquete *Crypto.Hash* se encuentra codificada la función hash que queremos calcular. Esto es por lo que solo será necesario crear una nueva clase y llamar a un método para obtener el resultado de esta función resumen, con entrada la variable data. El código es el siguiente:

```
funcion_hash = SHA256.new(data)
salida =funcion_hash.digest()
```

4.4 *Firmas digitales*

Como hemos mencionado previamente, se utilizará la firma digital RSA de la Figura 2-1 para identificar al emisor en el mensaje y evitar ataques de suplantación. Esto se realiza de una forma diferente en ambos lenguajes. En java existe la instancia "SHA256withRSA" que te calcula instantáneamente el proceso de realizar la función hash SHA-256 y la encriptación RSA con clave privada del mensaje original. En Python estas operaciones se harán por separado, como veremos a continuación.

4.4.1 Java

Utiliza la clase *Signature* del paquete *java.security*. Conseguimos realizar la firma de un mensaje (datos) utilizando la clave privada del actor (privateKey) con el siguiente código:

```
Signature firma = Signature.getInstance("SHA256withRSA");
firma.initSign(privateKey);
firma.update(datos);
byte[] salida=firma.sign();
```

4.4.2 Python

En Python, utilizaremos la clase *PKCS1_v1_5* del paquete *Crypto.Signature* para encriptar con la clave privada (privateKey) el digest de un mensaje (data) que calcularemos como en la sección 4.3.2.

Comenzamos obteniendo la clave privada como en la sección 4.2.2:

```
privateKey=RSA.importKey(key,'PEM')
```

Firmamos el mensaje:

```
firmar = PKCS1_v1_5.new(privateKey)
salida = firmar.sign(SHA256.new(data)
```

5 Pruebas, resultados y estudio

5.1 *Pruebas en el sistema*

Como a nivel de usuario las operaciones realizadas por los protocolos son transparentes, vamos a comenzar analizando el log de cada actor, imprimiendo en él las operaciones que realiza y los mensajes que recibe.

Protocolo 1

En el protocolo 1 se envían cuatro mensajes en total, dos entre Ui y SU y otros dos entre SU y M. Además, se realizan acciones de cifrado y firma RSA y generación de números aleatorios grandes. En la Figura 5-1 mostramos el log del actor SU durante este protocolo.

```
Se ha establecido una conexion con un actor
Leemos el mensaje recibido
Esperando para conectarse
Firma recibida correcta
Mensaje recibido de Ui: b'1;12345678A:secreto:'
Nos conectamos con M
Enviamos mensaje del protocolo a M
Mensaje recibido de M: b'3;1529458956:vI1KcFD2qAT16EbQAdB6Wz9zwJ89DLM3:'
Enviamos el mensaje de respuesta a Ui
```

Figura 5-1: Log SU Fase 1

Protocolo 2

El protocolo 2 contiene cuatro mensajes entre Ui y M. También se incluyen cálculos por parte de Ui sobre la generación de claves AES y generación de números aleatorios seguros, a la vez que ambos actores utilizan RSA y AES para cifrar sus mensajes. En la Figura 5-2 mostramos en Log de M en el que se incluye su parte realizada en esta fase en concreto.

```
Esperando para conectarse
Mensaje de SU recibido: b'2;336736097285853094074490099629659402719:'
Enviamos el mensaje de respuesta a SU
Cerramos la conexion con el cliente
Se ha establecido una conexion con un actor
Fase 2 del protocolo
Esperando para conectarse
Ui ha establecido conexion con nosotros
Mensaje de Ui recibido: b'5;vI1KcFD2qAT16EbQAdB6Wz9zwJ89DLM3:1529458956372:33673
6097285853094074490099629659402719:'
ValM correcto
Cliente correcto
Recibimos el segundo mensaje de Ui7;-8063859280199457002:1529458957.0675185:
Generamos la clave de acceso al sistema (ValSM) y la guardamos en la BD que cont
rola SM
Enviamos un mensaje a Ui dando por finalizada la fase 2 del protocolo
Cerramos la conexion con el cliente
Se ha establecido una conexion con un actor
SM ha establecido conexion con nosotros
Esperando para conectarse
Mensaje de SM recibido: b'13;Mensaje de peticion de token'
Protocolo 3. Peticion de token
Enviamos el token a SM
Cerramos la conexion con el cliente
```

Figura 5-2: Log M Fases 1, 2 y 3

Protocolo 3

En este protocolo se muestra el acceso al sistema. Son cinco mensajes entre Ui y SM más otro par de mensajes entre SM y M, solicitando este último una nueva clave de acceso a M. En la Figura 5-3 podemos observar el Log de SM durante toda la fase 3 del protocolo. En esta figura podemos destacar la clave de acceso al sistema, la cual se encuentra impresa por pantalla como un array de bytes después de la llegada del mensaje del protocolo con ID_Mensaje = 9.

```
omar@omar-Aspire-V3-571G:~/Escritorio/Server$ python SM.py
Levantamos el servidor SM. IP: localhost, puerto: 8000
Esperando para conectarse
Se ha establecido una conexion con un actor
ESTAMOS DENTRO
Esperando para conectarse
Fase 3 del protocolo
Mensaje de Ui recibido: b'9;1529458956410:'
b'\xd8+\x11Z!\xaf\x9f)\xbf|m\xfb\x192 \xd4F;S\x9e\x04\xbc\xd6\x9c\x8e\xb7\x1e\x9
5{\x0e\xce\xf2'
b'\xd8+\x11Z!\xaf\x9f)\xbf|m\xfb\x192 \xd4F;S\x9e\x04\xbc\xd6\x9c\x8e\xb7\x1e\x9
5{\x0e\xce\xf2'
Clave de acceso valida. El usuario anonimo tiene acceso a nuestro servicio
Recibimos el mensaje de Ui: 12;Peticion de logout:
El usuario no ha cometido ninguna ilegalidad en su sesion. Procedemos con el fin
al del protocolo 3
Enviamos mensaje del protocolo a M
Mensaje recibido de M: b'14;oU5z5ThivhzSBWQoIbD8vXQwizVe9D5G:'
Cerramos la conexion
```

Figura 5-3: Log SM Fase 3

Protocolo 4

Este protocolo realiza los pasos necesarios para la revocación de identidades. Cuenta con los cuatro actores, los cuales intercambian la información para llegar a revocar la anonimia de los usuarios que hayan infringido el contrato inicial. En el anexo R podemos encontrar la Figura 0-18, la cual muestra el mensaje que leerán los usuarios cuya anonimia sea revocada.

```
06-20 03:42:19.398 9874-9903/com.example.root.damas D/OpenGLRenderer: Swap behavior 0
06-20 03:42:31.194 9874-9874/com.example.root.damas W/IInputConnectionWrapper: finishComposingText on inactive InputConnection
06-20 03:42:31.194 9874-9874/com.example.root.damas W/IInputConnectionWrapper: finishComposingText on inactive InputConnection
06-20 03:42:36.196 9874-10157/com.example.root.damas D/PROTOCOLO 1: EMPEZAMOS LA FASE 1 DEL PROTOCOLO
06-20 03:42:36.199 9874-10157/com.example.root.damas D/PROTOCOLO 1: Conexion establecida con SU
06-20 03:42:36.199 9874-10157/com.example.root.damas D/PROTOCOLO 1: Enviamos el mensaje de registro de la fase 1 del protocolo
06-20 03:42:36.264 9874-10157/com.example.root.damas D/PROTOCOLO 1: Recibimos de SU: 4;336736097285853094074490099629659402719:1529458956:vI1KcFD2qAT16EbQAdB6Wz9zwJ89DLM3
06-20 03:42:36.264 9874-10157/com.example.root.damas D/PROTOCOLO 1: FIN DE LA FASE 1 DEL PROTOCOLO 4;336736097285853094074490099629659402719:1529458956:vI1KcFD2qAT16EbQAdB6Wz9zwJ89DLM3
06-20 03:42:36.322 9874-9874/com.example.root.damas W/art: Before Android 4.1, method android.graphics.PorterDuffColorFilter android.support.graphics.drawable.VectorDrawableCompat.updateTi
06-20 03:42:36.371 9874-10165/com.example.root.damas D/PROTOCOLO 2: EMPEZAMOS LA FASE 2 DEL PROTOCOLO
06-20 03:42:36.372 9874-10165/com.example.root.damas D/PROTOCOLO 1: Conexion establecida con SU
06-20 03:42:36.372 9874-10165/com.example.root.damas D/PROTOCOLO 2: Enviamos el mensaje de acceso a M de la fase 2 del protocolo
06-20 03:42:36.389 9874-10165/com.example.root.damas D/PROTOCOLO 2: Recibimos de M: 6;15294589563721529458957.0675185
06-20 03:42:36.389 9874-10165/com.example.root.damas D/PROTOCOLO 2: Variable salt creada: -8063859280199457002
06-20 03:42:36.391 9874-10165/com.example.root.damas D/PROTOCOLO 2: Recibimos de M 8;OK:
06-20 03:42:36.391 9874-10165/com.example.root.damas D/PROTOCOLO 2: FIN DE LA FASE 2 DEL PROTOCOLO
06-20 03:42:36.408 9874-10166/com.example.root.damas D/PROTOCOLO 3: EMPEZAMOS LA FASE 3 DEL PROTOCOLO
06-20 03:42:36.410 9874-10166/com.example.root.damas D/PROTOCOLO 2: Enviamos el mensaje de acceso a SM de la fase 3 del protocolo
06-20 03:42:36.425 9874-10166/com.example.root.damas D/PROTOCOLO 3: Recibimos de SM: 10;1529458956410:1529458957.1032765
06-20 03:42:36.427 9874-10166/com.example.root.damas D/PROTOCOLO 3: Recibimos de SM 15;Acceso correcto
06-20 03:42:36.427 9874-10166/com.example.root.damas D/PROTOCOLO 3: HEMOS ACCEDIDO CORRECTAMENTE AL SERVICIO
06-20 03:42:36.427 9874-10166/com.example.root.damas D/PROTOCOLO 3/4: PETICION DE DESCONEXION AL SISTEMA
06-20 03:42:36.427 9874-10166/com.example.root.damas D/PROTOCOLO 3/4: Enviamos el mensaje de peticion de logout a SM de la fase 3/4 del protocolo
06-20 03:42:36.454 9874-10166/com.example.root.damas D/PROTOCOLO 3/4: Recibimos de SM: 15;oU5z5ThivhzSBWQoIbD8vXQwizVe9D5G
06-20 03:42:36.454 9874-10166/com.example.root.damas D/PROTOCOLO 3: NOS HEMOS DESCONECTADO CORRECTAMENTE DEL SERVICIO
06-20 03:42:36.463 9874-9874/com.example.root.damas E/RecyclerView: No adapter attached; skipping layout
06-20 03:42:36.474 9874-9874/com.example.root.damas W/art: Before Android 4.1, method int android.support.v7.widget.ListViewCompat.lookForSelectablePosition(int, boolean) would have incorr
```

Figura 5-4 : Log Ui Fases 1, 2 y 3

En la Figura 5-4 mostramos el Log de Ui durante todas las fases del protocolo, hasta desconectarse del servicio. Como hemos podido ver en esta sección, los mensajes que se transmiten los actores son moderadamente pequeños (256 bytes con firma RSA), por lo que no va a suponer un problema a nivel de congestión de red.

5.2 *Tiempos de respuesta*

Debido al **RNF9** (ver sección 3.2.2), el tiempo de respuesta del sistema es un aspecto que analizar a la hora de evaluar el funcionamiento de los protocolos.
Para ello, incluimos una tabla que indica el tiempo de respuesta del sistema en ejecutar determinadas acciones bajo ciertas condiciones:

Fases / Usuarios registrados	Fase 1	Fase 2	Fase 3	Fase 4	Acceso	Creación clave acceso
10	0,087	0,040	0,076	0,080	0,203	0,001
100	0,088	0,045	0,080	0,085	0,218	0,001
1000	0,102	0,062	0,099	0,090	0,263	0,001
10000	0,120	0,290	0,140	0,150	0,550	0,001
100000	0,140	0,576	0,220	0,430	0,936	0,001

Tabla 5-1: Tiempos de respuesta

La Tabla 5-1 incluye la información media del tiempo de respuesta del sistema en segundos de acuerdo con el número de clientes registrados. A medida que este número aumenta, las bases de datos del sistema tienen mayor peso, por lo que se requiere más tiempo para realizar el acceso. También se podría analizar el tiempo de respuesta bajo una multitud de conexiones simultáneas al servidor, pero estos datos estarían envenenados ya que en la prueba de concepto del protocolo de anonimia diseñado en 3.1, todos los actores concurren en un mismo PC. Luego ese análisis de rendimiento dependería mayoritariamente de las especificaciones del PC y no del protocolo, que es lo que queremos estudiar.

5.3 *Requisitos computacionales*

Para realizar las pruebas de este TFG se ha usado un único ordenador con un procesador Intel Core i7-3630QM de 2.4 GHz y 8 GB de DDR3 RAM. La parte del cliente se ha simulado en un emulador de Nexus 7.

De acuerdo con los resultados de 5.1, no se utilizan muchos datos en los mensajes del protocolo ya que los mensajes suelen ser de 256 bytes. El punto crítico dependerá de la información que necesite SM para proporcionar el servicio requerido.
Respecto a los resultados de 5.2, el tiempo de respuesta cumple con el requisito propuesto en el **RNF9**. No obstante, no se ha estudiado la variación del tiempo de respuesta del servicio frente a un número elevado de peticiones. En ese caso, nuestra estructura del sistema de la sección 3.1 no ofrece mucha capacidad, por lo que para que el sistema pueda mejorarse y adaptarse de manera fluida y sin perder calidad ante este caso se ha realizado un estudio de la escalabilidad del sistema que veremos en la siguiente sección.

5.4 *Estudio de escalabilidad*

En un primer diseño del sistema, solo hay un servidor que cumple el papel de los tres actores, conectados a través de localhost.
Esto supone un problema en caso de que un actor caiga y no pueda realizar su funcionalidad, ya que no hay forma de continuar el correcto funcionamiento. También influye el hecho de que es un mismo ordenador el que ejecuta todas las operaciones, por lo que tendrá problemas de saturación del sistema a partir de ciertas conexiones simultáneas.

Una buena solución a este problema es la posibilidad de tener varios servidores alrededor del mundo que realicen las tareas de cada actor. Con esta vista, el sistema sería capaz de tolerar caídas de alguno de los servidores, siempre y cuando haya una buena redirección de los mensajes de los clientes a través de los servidores oportunos activos. Además, al dividir las operaciones en distintos ordenadores, la capacidad del sistema para manejar un mayor número de conexiones simultáneas sin sufrir saturación aumenta.
Pero para poder concurrir a múltiples servidores habría que tener en cuenta el acceso y modificación de las bases de datos usadas por estos, además de la duplicación de estas en varios lugares y su correcta actualización.

En resumen, un cambio a un sistema descentralizado (la Figura 5-5 muestra un grafo con las diferencias entre un sistema centralizado (A) y otro descentralizado (B)) soluciona el problema de un único servidor con recursos finitos al que se conectan todos los clientes. Esto también implica un cambio de gestión de las bases de datos a un modelo descentralizado, lo cual añade problemas de gran envío de información entre los servidores para mantener las bases de datos actualizadas en todo momento. Blockchain permisivo (ver sección 2.8) puede evitar estos problemas ya que toda la información necesaria se guardaría en la red y cualquier actor con privilegios de acceso no podrá modificar datos sin el consenso del resto de nodos participantes.

Por último, se necesitaría analizar la forma en la que los clientes se conectan a un determinado servidor y no a otro cuando quieran realizar una conexión. Para ello tendríamos en cuenta (y siguiendo un orden descendiente de prioridad):

i. Disponibilidad del servidor: A mayor número de peticiones realizadas simultáneamente, menor disponibilidad a la hora de obtener más conexiones.
ii. Distribución geográfica del servidor: Una mayor cercanía geográfica entre el cliente y el servidor disminuye el tiempo de espera de envío de mensajes.

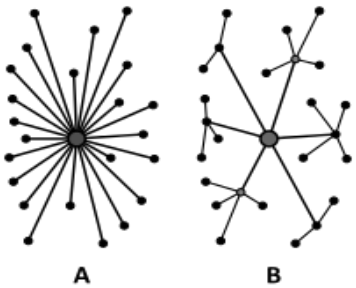

Figura 5-5: Sistema centralizado (A) vs descentralizado (B)[4]

[4] Fuente: Wikidepia. Autor: Kes47

5.5 *Estudio de seguridad*

En esta sección analizamos unos ejemplos de posibles ataques que podrían realizarse al sistema y la respuesta de este.

5.5.1 Manipulación de mensajes

Uno de los ataques más conocidos es el de "man-in-the-middle" (anexo S). En este caso, una tercera persona podría tener acceso a los mensajes enviados, además de poder leer y o manipular a su gusto.

El caso de leer estos mensajes no supone ningún problema, ya que todos los mensajes irán cifrados. Si van cifrados con RSA, para poder leer el mensaje, el atacante necesitaría la clave privada del receptor, la cual suponemos que guarda bajo seguridad. Si van cifrados con AES, necesitaría conocer la clave que previamente han intercambiado las entidades. También le sería imposible, porque esa clave se encuentra cifrada mediante RSA.

En el caso particular de manipulación de mensajes, si estos son cifrados por RSA, se podría generar un nuevo mensaje con el que sustituir el mensaje original del emisor, ya que sólo se necesita conocer la clave pública del receptor para cifrar los mensajes. La solución a este problema implementada en este trabajo es la incorporación de las firmas RSA (ver Figura 2-1). El atacante no podría generar la firma del mensaje que intenta modificar ya que desconoce la clave privada del emisor, por lo que el receptor se daría cuenta de que el mensaje ha sido modificado.

5.5.2 Replica de mensajes

Otro ataque muy simple es replicar un mensaje correcto ya enviado previamente por un usuario para hacerse pasar con él. También sería imposible, porque se añaden marcas temporales en ciertos mensajes. Si un atacante intenta atacar al sistema mediante la replicación de mensajes correctos que usuarios validos han inercambiado con el servidor previamente, el sistema se daría cuenta de este desfase de tiempo por las marcas temporales y el atacante no conseguiría su objetivo.

5.5.3 Suplantación de actores

La suplantación de un actor por parte de un tercero supondría que el servicio se interrumpiría, ya que impediría las acciones del actor suplantado. En la sección 5.4, se menciona una solución a este problema llegando a un sistema distribuido.

Pero, aunque un tercero llegue a suplantar a un actor del sistema, este no tendría acceso ni a las bases de datos ni al contenido de los mensajes enviados (sigue necesitando la clave privada del actor original), así que no podría realizar ninguna acción en el sistema.

5.5.4 Suplantación de identidad de un usuario

Según el diseño, los clientes contienen información sobre su ID usado en el sistema ya que estos son enviados al cliente en el protocolo de la sección 3.3.1. Esto puede generar ataques de suplantación de identidades de usuario al conocer la información de cómo son estos identificadores y poder deducir otro ID válido del sistema.

Como podemos apreciar en el protocolo de la sección 3.3.2, M no se encarga de comprobar que el ID enviado por el cliente es el que corresponde, así que si un atacante con un token de acceso a M válido podría hacerse pasarse por otro cliente si consigue de alguna manera su ID. Pero debido a la forma de creación de las claves de acceso de la sección 3.3.2, el atacante no logrará producir la correcta clave de acceso para ese usuario al no tener todas las entradas necesarias. Además, no podrá obtener otro token de acceso a M ya que este sólo se consigue en el protocolo de la sección 3.3.3.

5.5.5 Trazabilidad del sistema

Se puede intentar trazar las acciones de los usuarios para obtener información sobre ellos, o al menos tener indicios de las acciones que determina un cierto usuario.
Un ejemplo muy sencillo sería que, si solo hubiera un cliente en el sistema, cualquier tercera persona que vea los mensajes que se envían a los servidores va a saber con certeza que proceden del mismo usuario.
Una solución de diseño para evitar la trazabilidad de las acciones de los usuarios ha sido la división de los protocolos de las secciones 3.3.2 y 3.3.3 en dos fases diferentes que pueden ser ejecutados en cualquier momento. Aunque es cierto que la fase 2 tiene que ser ejecutada siempre como paso previo a la fase 3, no tiene que ser inmediatamente antes, ya que se podría analizar los accesos a los servidores y llegar a la conclusión que ciertos mensajes proceden de un mismo usuario debido a su secuencia.
También se ha diseñado la función generadora de claves de forma que no se pueda trazar a los usuarios reales del servicio, siguiendo los pasos de la sección 3.7.

5.5.6 Inyección SQL

Debido al manejo de bases de datos, es necesario prevenir ataques de inyección SQL (anexo S) para evitar las operaciones no permitas sobre nuestras bases de datos.
La solución a este tipo de ataques es incluir una serie de filtros en la lectura de los mensajes.

5.5.7 Ataques DDOS

Un ataque de denegación de servicio (anexo S) podría causar que el servidor sea inaccesible por los clientes, por lo que impediría el funcionamiento del sistema. Una solución a este tipo de ataques es un sistema distribuido de servidores, donde la caía de un servidor no afecta a la caída total del sistema.

5.5.8 Robo de datos

Si un atacante consigue hacerse con ciertos datos, podría hacerse pasar por clientes válidos del sistema y realizar acciones no permitidas sin ser descubierto. Por ello es necesario la protección de los datos por parte de todas las partes. Una solución a este problema podría ser guardar los datos cifrados.
Analizando las variables que tienen lugar en el diseño del sistema, para que un atacante pueda hacerse pasar por un usuario debería robarle toda la información conseguida en la fase 1 de la sección 3.3.1. Esto es, necesitaría conocer el ID, T y ValM ya que esas tres variables serán las usadas a partir de ese momento como identificación del usuario. Otra medida para mejorar el sistema respecto a esta amenaza sería añadir la funcionalidad del sistema para requerir actualizar estas variables cada cierto tiempo.

5.5.9 Fuerza bruta

Una posible vulnerabilidad del sistema sería ataques por fuerza bruta, en los que se intente conseguir una clave de acceso válida para entrar al sistema, suplantando a un usuario.
Para prevenir este tipo de ataques hay que tener en cuenta la probabilidad matemática de producir una clave válida. En nuestro sistema, usamos claves de 256 bits (ver sección 3.7) que son frecuentemente actualizadas debido al acceso de los clientes. Para prevenir este tipo de ataques, se podría añadir una fecha de caducidad a las claves, dando lugar a que el cliente tuviera que pedir claves de acceso nuevas en caso de que estas no valgan.

5.6 *Estudio del nivel de anonimia y privacidad alcanzados*

Como explicamos anteriormente en la sección 1.2, el objetivo principal de este trabajo es la capacidad de otorgar un anonimato justo a los usuarios válidos de una determinada aplicación o web.

Según el capítulo 2.5.2, se pueden distinguir dos tipos de anonimato, respecto a las comunicaciones y respecto a la información.
En este trabajo se centra en conseguir el anonimato en la información. Si se deseara obtener también el anonimato en las comunicaciones, sería necesario el intercambio de información del protocolo explicado en la sección 3.3 bajo una red segura, como podría ser TOR.
Pero el anonimato en la información alcanzado no es total ya que, en la primera fase del protocolo se incluye información real por parte del cliente para poder identificarse. Esto es debido a que en nuestro sistema se incluye la posibilidad de poder revocar el anonimato a los usuarios que cometan acciones ilegales. Por lo que, el servicio que ofrece nuestro sistema no será el de garantizar el anonimato total a los usuarios, sino que propone un anonimato justo de acuerdo con el explicado en el capítulo 2.5.6.

La manera de llegar a estas condiciones de anonimato justo en nuestro trabajo es bajo la división de información. El servidor del sistema se encuentra dividido en varios actores independientes que van almacenando datos de los clientes a lo largo del proceso. Pero estos datos a vista de cada actor no contienen ninguna información completa por parte de los clientes del sistema, sino que es necesario juntar toda esta información para poder conseguir la identificación real de los clientes. Este proceso de juntar la información se realiza únicamente bajo el supuesto caso de que el cliente incumpla de alguna manera el contrato firmado con el que se registró en el sistema, y es el denominado proceso de revocación de identidades.
La división de información que realiza nuestro sistema es la siguiente.
SU se encarga únicamente de dar acceso a los clientes válidos del sistema asignándoles un ID y proporcionándoles una marca temporal T y un token de acceso a M, ValM. Pero SU no vuelve a tener conocimiento acerca de lo que el cliente hace más adelante en el sistema.
M proporciona claves de acceso a partir del ID que le envían los usuarios que posean un token para poder acceder a él. M en ningún momento llega a conocer la identidad real de los usuarios, ni siquiera si estos dicen la verdad y son el ID que realmente dicen ser. Tampoco tiene ninguna información sobre la entrada al servicio por parte de los usuarios.
SM permite la entrada a los usuarios que le aporten una clave válida. En esta posición los usuarios son anónimos, ya que irán accediendo con claves totalmente distinta a SM sin que él pueda determinar quién entra en el servicio concretamente.

Por último, en cuanto a la privacidad, como los usuarios acceden de manera anónima al servicio, este no puede guardar la información individual de estos usuarios como tales, aunque si puede sacar estadísticas conjuntas sobre datos de su sistema. Un ejemplo sencillo de este caso sería el de una aplicación de venta por internet. Bajo el nivel de anonimato conseguido, el sistema no puede almacenar la información de los productos que va comprando cada cliente, pero puede ver los productos más solicitados para ayudarle a mejorar su negocio y usar estos datos para hacer recomendaciones globales.

Para concluir, podemos comparar nuestro estudio sobre el anonimato justo (ver sección 2.5.6) con otros trabajos actuales, los cuales se basan en proporcionar distintos niveles de anonimia según el nivel de anonimia requerido por cada usuario [25].

6 Conclusiones y trabajo futuro

6.1 *Conclusiones*

En este trabajo hemos estudiado y analizado un protocolo criptográfico de anonimato y privacidad justa. Para ello, comenzamos viendo que un sistema con anonimia total en la red sería contraproducente, ya que, bajo esa suposición sería imposible identificar a los usuarios que cometan ilegalidades. Entonces, el objetivo principal a conseguir en este TFG mediante el protocolo propuesto es poder otorgar el anonimato justo a todos los usuarios válidos del sistema que se comporten de acuerdo con un contrato previamente acordado. Pero, únicamente en el caso de que algún usuario infrinja el contrato, el sistema podrá revocar su anonimato, sin llegar a afectar al resto de usuarios válidos ni al funcionamiento del servicio.

La forma de obtener la anonimia con el protocolo es mediante la división de información entre los actores que forman el sistema, que son cuatro. El cliente (Ui), y tres servidores (SU, M y SM) que se encargarán de realizar funcionalidades y gestionar información diferente de manera independiente.
Además, el protocolo se dividirá en cuatro fases, en las cuales se irá aumentando el nivel de anonimia del cliente en el sistema, mientras que la última fase solo se ejecuta en el caso de revocación de la anonimia. En la primera fase se cambia la información real de los usuarios por un pseudónimo (su ID). En la segunda fase se eliminará la trazabilidad generada por usar pseudónimos y se otorgará a los clientes claves de acceso únicas para cada sesión que deseen realizar. En la tercera fase se gestiona el acceso anónimo de los usuarios que aporten una clave de acceso válida.

Una vez propuesto e implementado el protocolo, se ha llegado a analizar y a estudiar el nivel de anonimia conseguido junto a un análisis de seguridad y rendimiento del sistema. Para ello se ha probado el protocolo como método de generar anonimia a los clientes de un foro anónimo. Podemos concluir que, como prueba de concepto, nuestro protocolo cumple con el objetivo de conseguir la anonimia y privacidad justa, siempre y cuando contemos con la hipótesis de que los actores del sistema se comportan de manera correcta.

6.2 *Trabajo futuro*

Este trabajo deja planteados unos posibles retos futuros:

- Dividir a los actores del protocolo en múltiples ordenadores y analizar su rendimiento.
- Realizar una publicación científica acerca del protocolo diseñado y analizar posibles adaptaciones o mejoras.
- Probar el protocolo de anonimia bajo un servicio con una importante actividad.
- Estudiar el tiempo de validez que el sistema proporcionará a las claves de acceso e implementar la funcionalidad para actualizarlas automáticamente.
- Implementar la funcionalidad de Blockchain permisivo.

Referencias

[1] Auguste Kerclhoffs, “La cryptographie militaire”, Journal des sciences militaires, vol IX, pp. 5-83, 1883

[2] J. Quisquater, Louis C. Guillou, Thomas A. “How to Explain Zero-Knowledge Protocols to Your Childen”, Springer-Verlag, 1990, pp. 628-631

[3] D. Chaum, E. van Heyst “Group signatures”, EUROCRYPT’91, pp. 257-265

[4] D. Chaum. “Blind signature systems. In advances in Cryptology”, CRYPTO ‘83, 1984, pp. 153

[5] OpenSSL Software Foundation, “OpenSSL”, 2018, https://www.openssl.org/

[6] OpenSSL Software Foundation, “Man pages for 1.1.1”, 2018, https://www.openssl.org/docs/man1.1.1/

[7] “Java Cryptography Architecture (JCA) Reference Guide”, 2018, https://docs.oracle.com/javase/7/docs/technotes/guides/security/crypto/CryptoSpec.html

[8] “REGLAMENTO (UE) 2016/679 DEL PARLAMENTO EUROPEO Y DEL CONSEJO”, Julio 2016, https://www.boe.es/doue/2016/119/L00001-00088.pdf

[9] M. Artin, “Algebra”, Prentice Hall, 1992

[10] W. Diffie, M.E. Hellman, “New directions in cryptography”, IEE Transactions on Information Theory 22, 1976, pp. 644-654

[11] Niven, Zukerman, “Teoría de los números”, 1985, pp. 69

[12] R.L. Rivest, A. Shamir, L. Adleman. “A method for Obtaining Digital Signatures and Public-Key Cryptosystems”, 1978

[13] NIST-FIPS 197. “Avanced Encryption Standard (AES)”

[14] William Stallings. “Cryptography and Network Security: Principles and Practice”, Prentice Hall, 4th Edition, 2005

[15] J. H. Silverman. “The aritmetic of Elliptic Curves”, Springer-Verlag, 1986

[16] S. Lang. “Elliptic Functions”, Addison-Wesley, 1973

[17] M. Bellare, P. Rogaway. “Optimal asymmetric encryption – How to encrypt with RSA”, EUROCRYPT ’94, 1995, pp. 90-111

[18] D. Chaum, “Demonstrating that a Public Predicate Can be Satisfied Without Revealing Any Information about How”, CRYPTO 86’, Springer-Verlag, 1987, pp.195-199

[19] A. Kiayias, Y. Tsiounis, M. Yung. “Traceable Signatures”, Springer Berlin Heidelberg, 2004, pp. 571-589

[20] D. Boneh, X. Boyen, H. Shacham. “Short Group Signatures”, Springer Berlin Heidelberg, 2004, pp. 41-55

[21] E. Wall, G. Malm. “Using Blockchain Technology and Smart Contracts to Creare a Distributed Securities Depository”, Department of Electrical and Information Technology, Lund University, 2016

[22] Lance Koonce, “Let’s Disintermediate All the Lawyers: Smart Contracts on the Blockchain (Why Blockchain Matters to the Arts, Part 4)”, 2016

[23] Antony Lewis, “A gentle introduction to blockchain technology”, BraveNewCoin

[24] Vimi Grewal-Carr, Stephen Marshall, “BLOCKCHAIN. ENIGMA. PARADOX. OPORTUNITY”, Deloitte, 2016

[25] J. Diaz, S. G. Choi, D. Arroyo, Angelos D. Keromytis, Francisco B. Rodriguez, M. Yung. “Privacy in e-Shopping Transactions: Exploring and Addressing the Trade-Offs”, Lecture Notes in Computer Science, vol 10879, Springer, 2018

Glosario

AES	Advanced Encryption Standard
API	Application Programming Interface
CS	Cryptosystem
DES	Data Encryption Standard
DSA	Digital Signature Algorithm
ECC	Elliptic Curve Cryptography
IDEA	International Dara Encryption Algorithm
JCA	Java Cryptography Architecture
KPG	Key Pair Generator
M	Moderator
MAC	Message Authentication Code
MDC	Message Detection Code
RSA	Public Key Cryptosystem
SHA	Secure Hash Algorithm
SC	Smart Contract
SM	Service Manager
SQL	Structure Query Language
SSL	Security Socket Layer
STL	Security Transport Layer
SU	Sign Up
TOR	The Onion Router
Ui	User Interface
ZKP	Zero Knowledge Proof

Anexos

A Manual de instalación

En este TFG hemos usado dos lenguajes de programación, Python y Java sobre un sistema operativo Ubuntu 14.4.

En Python, para poder trabajar con el sistema es necesario tener la última versión instalada. En nuestro caso será la versión 3.5 que se instala con los siguientes comandos.

$ sudo apt-get install python 3.5

Para poder usar las funciones criptográficas de la sección 4, hay que tener instalada la librería pycrypto. Se instala con los siguientes comandos.

$ pip install pycrypto

En el caso de Java, hemos usado el programa Android Studio, el cual ofrece también varias máquinas virtuales para poder probar nuestra aplicación. Para instalar este programa hay que descargarse un fichero comprimido desde su página web como muestra la Figura 0-1 y después seguir una serie de pasos.

Android Studio downloads

Platform	Android Studio package	Size	SHA-256 checksum
Windows (64-bit)	android-studio-ide-173.4819257-windows.exe Recommended	758 MB	2d11cd16ffefc7f4aca82bd95b0d0ca849a854a07ba1a353adf65dfc102aee9b
	android-studio-ide-173.4819257-windows.zip No .exe installer	855 MB	6941761a9324998d9cdf5d7548ff16fcb65c9e71ea70e1bc75d066b51b77c7dd
Windows (32-bit)	android-studio-ide-173.4819257-windows32.zip No .exe installer	854 MB	85cbfb33c94183abcb70caac34ea034214a079376fb1b2e10bc7b4ed71c05cf0
Mac	android-studio-ide-173.4819257-mac.dmg	855 MB	21ec7cf480bfa05ff90594e9cebd0f79892e41d37b4a14988dce04a6fbb76b58
Linux	android-studio-ide-173.4819257-linux.zip	853 MB	d86748e44d658fd39581b40f7b706fb397fc1eca5dd6f8066a56c0beb856dea4

Figura 0-1: Descarga Android Studio

1. Una vez descargado el archivo deseado, hay que descomprimirlo en la ruta en la que se desea realizar las aplicaciones.
2. Para iniciar Android Studio, simplemente hay que navegar dentro de la carpeta *android-studio/bin/* y ejecutar *studio.sh*.
3. Se preguntará si se desea o no importar las configuraciones previas de Android Studio. El usuario elegirá la opción que prefiera.
4. El proceso continuará por el asistente de configuración de Android Studio, el cual servirá de guía para el usuario. En este paso se incluye la descarga de los componentes Android SDK que el usuario quiera incluir en sus aplicaciones.

B Manual del programador

La ejecución del código implementado es muy simple, ya que debido al requisito no funcional **RNF8** de la sección 3.2.2, el protocolo criptográfico para conseguir la anonimia es transparente para los usuarios.
Para poder llevar un seguimiento de las acciones de nuestros actores definidos en la sección 3.1, es necesario atender a los mensajes que se irán imprimiendo en el log de cada uno. Entonces, un programador solo necesitará lanzar los servidores y abrir la máquina virtual que simula al cliente para tener el sistema en funcionamiento. Después, las acciones que realizará nuestro sistema según las operaciones realizadas a través de la interfaz del cliente se irán mostrando en el log.

Lanzar los servidores:
Lanzar los servidores solo requiere ejecutar tres programas Python como mostramos a continuación en la **¡Error! No se encuentra el origen de la referencia.**.

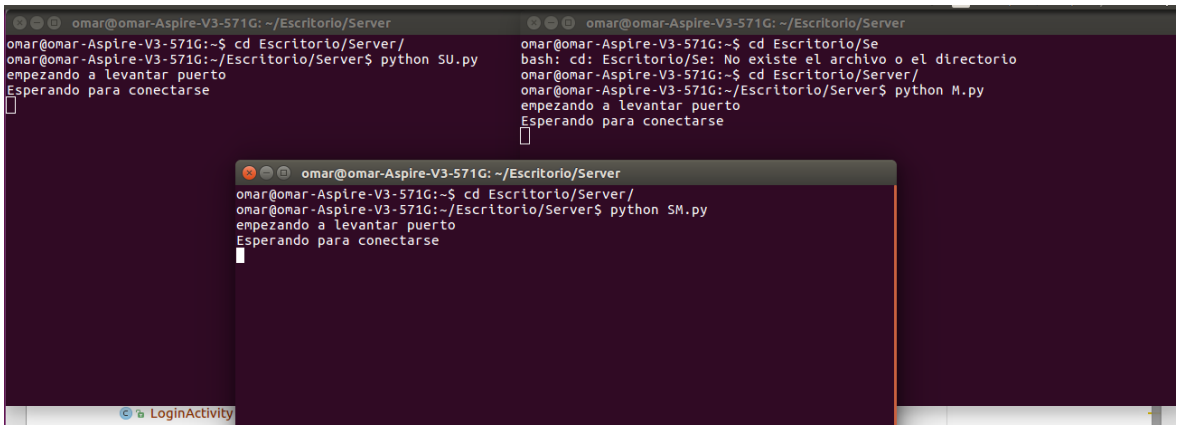

Figura 0-2: Lanzar los servidores

Abrir máquina virtual para ejecutar el cliente.
Para ello hemos usado una máquina virtual de Android Studio que simula el comportamiento de nuestra aplicación en un dispositivo móvil Nexus 7. Para ello hay que realizar los siguientes pasos:

1. Ejecutar la aplicación, como muestra la **¡Error! No se encuentra el origen de la r eferencia.**:

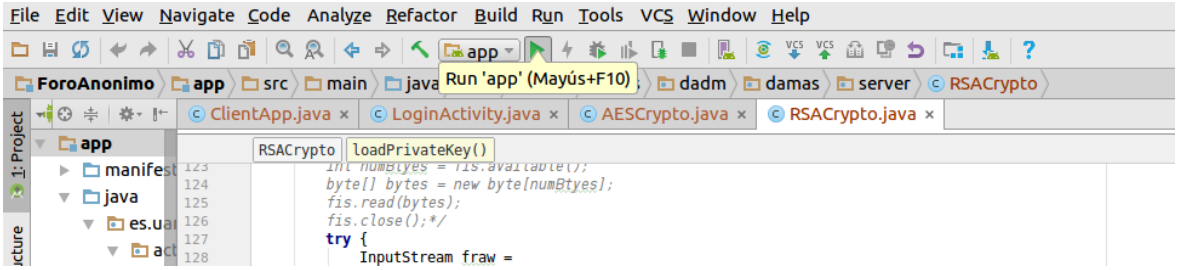

Figura 0-3: Run App

2. Elegir la Máquina Virtual (MV) en la que se simulará (ver Figura 0-4):

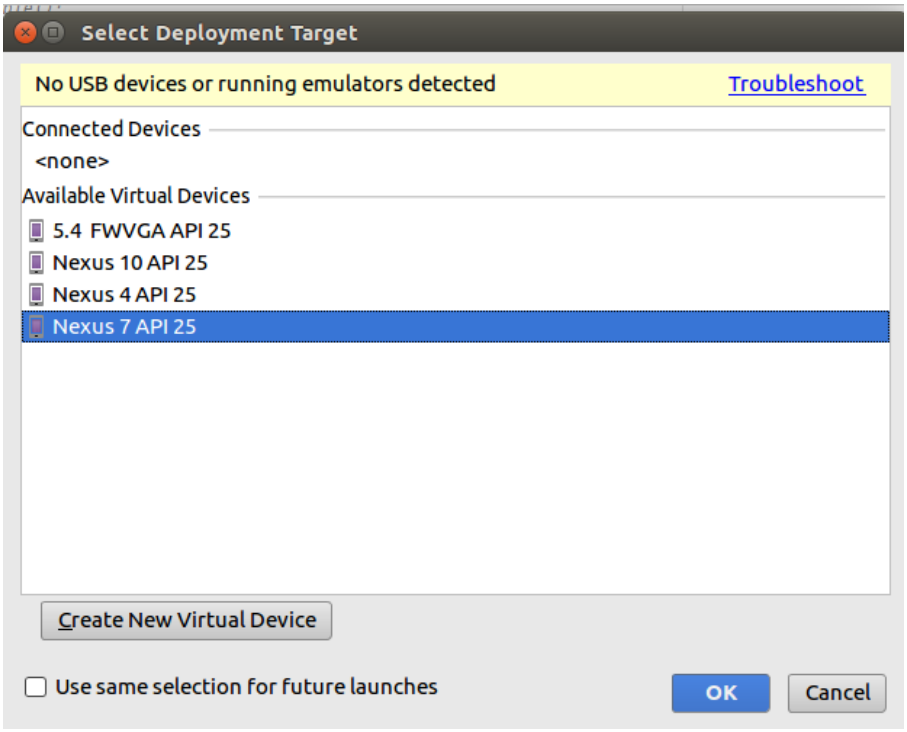

Figura 0-4: Selección de la MV

Si se desea mirar los mensajes escritos por este actor, hay que atender a la sección de Andorid Monitor de la Figura 0-5 que se encuentra dentro de la aplicación Android Studio.

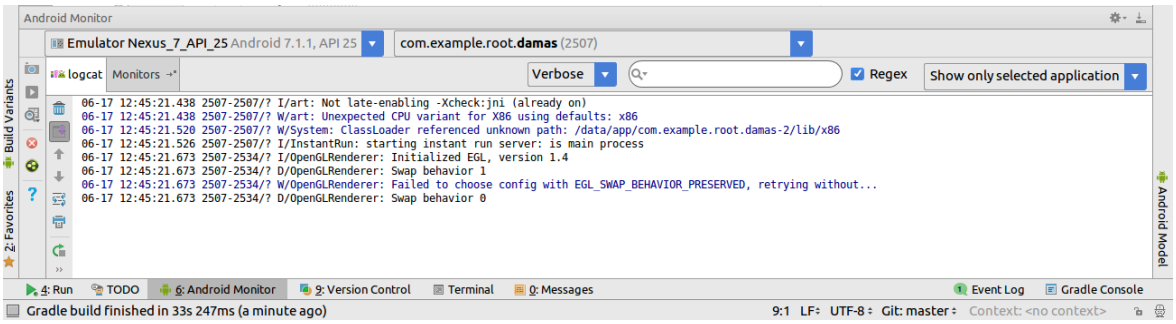

Figura 0-5: Android Monitor

C Historia de la criptografía

La criptografía surgió como la necesidad para ocultar mensajes. Para evitar que sus enemigos consiguieran información importante cuando lograban capturar a un mensajero, las primeras civilizaciones idearon técnicas de ocultación de mensajes que podemos considerar como el origen de la criptografía.

El primer método de criptografía documentado fue la "Escítala" Espartana en el siglo V A.C. Consistía en enrollar el mensaje en un cilindro que servía como clave para lograr la transposición de las letras y así poder encriptar y desencriptar el mensaje.

El segundo método documentado se atribuye al historiador griego Polibio, el cual consistía en la sustitución de las letras del texto por números, basándose en la posición en la que se encontraban las letras en una tabla.

Otro método famoso antiguo de criptografía es el Criptosistema (anexo D) César, también basado en la sustitución. Cada letra del mensaje se cambia por la letra que ocupaba una determinada posición en el alfabeto. En el caso concreto del Cifrado César, las letras se cambian por las que ocupan tres posiciones más adelante. Es decir, la letra A (posición 1 del alfabeto) se sustituiría por la letra D (posición 4) y, por ejemplo, la letra Z (última posición del alfabeto) se sustituía por la letra C (posición 3), es decir, volviendo a contar desde la primera posición al llegar a la última posición del alfabeto, llegando así a formar un ciclo.
Utilizando notación matemática, podríamos considerar las posiciones del alfabeto como un grupo algebraico (anexo E), $\mathbb{Z}_{27}$.
Sea n el tamaño del alfabeto. Sea k $\mathbb{Z}_{27}$ el número de posiciones que tenemos que avanzar para cambiar las letras. Sea x la posición que ocupa una determinada letra. Sean $f_k(x)$ y $f_k^{-1}(x)$ dos funciones sobre $\mathbb{Z}_{27}$ tales que:

$$f_k(x) = x + k$$
$$f_k^{-1}(x) = x - k$$

Según esta notación, las dos funciones servirían para cifrar y descifrar una letra, respectivamente. En el caso anterior, n sería el tamaño del alfabeto y k=3. Se observa que es necesario conocer una determinada información (en el caso de la Escítala, el diámetro del cilindro; en el caso del criptosistema de Polibio, la tabla y en el caso del Criptosistema Cesar, la posición k) la cual denominaremos clave. Una tercera persona que no sepa la clave no podrá entender el mensaje encriptado salvo que logre romper el Criptosistema o el tamaño de las posibles claves sea muy pequeño para llegar a obtener la clave a base de ir probando todas las claves posibles.
Por ejemplo, el tamaño de clave del Criptosistema César es el número de letras del alfabeto, es decir 27. Como es un tamaño de clave muy pequeño, ha sido necesario introducir mayor complejidad en el Criptosistema para aumentar el tamaño de clave y conseguir que sea imposible que una tercera persona llegase a conseguir la clave mediante fuerza bruta.

Por esta razón, los Criptosistemas han ido evolucionando hasta llegar a métodos de cifrados ya no tan sencillos como puede ser la Máquina Enigma. La Máquina Enigma fue un criptosistema mecánico usado por las tropas alemanas para transmitir mensajes militares durante la Segunda Guerra Mundial.

Otra razón por la que los CS han ido aumentando la complejidad es debido a la adaptación y a las necesidades de la época en la que se usaban. Hoy en día, gracias a los ordenadores podemos realizar multitud de cálculos en muy poco. Esto ha llevado a los CS a seguir evolucionando hasta los que utilizamos en la actualidad, o como podemos denominarlos, los métodos de cifrado moderno.

D Criptosistema

Definición: Criptosistema
Un criptosistema es una 5-tupla (P, C, K, E, D) que cumple las siguientes condiciones:

1. P es un conjunto finito de posibles textos planos.
2. C es un conjunto finito de posibles textos cifrados.
3. K es un conjunto finito de posibles claves.
4. Para cada $k \epsilon K$ hay una regla de encriptación $e_k \epsilon E$ y una regla de desencriptación $d_k \epsilon D$.
5. Cada $e_k : P \to C$ y $d_k : P \to C$ son dos funciones que cumplen $d_k(e_k(x)) = x \ \forall x \in P$.

Ejemplos:
A continuación, mostramos una serie de ejemplos sobre varios criptosistemas famosos que servirán como punto de partida para introducir el concepto de criptosistema.

- **Cifrado de desplazamiento**

$P = K = C = \mathbb{Z}_{27}$
$Para\ cualquier\ k: 0 \leq k \leq 27\ definimos$
$e_k(x) = x + k\ mod\ 27$
$d_k(y) = y - k\ mod\ 27$
$donde\ x\ e\ y\ \in\ \mathbb{Z}_{27}$

- **Cifrado de sustitución**

$P = C = \mathbb{Z}_{27}$
$K\ son\ las\ permutaciones\ de\ \{0, \dots, 26\}$
$Para\ cualquier\ permutación\ \pi \in K\ definimos$
$e_\pi(x) = \pi(x)$
$d_\pi(y) = \pi^{-1}(y)$
$donde\ x\ e\ y\ \in\ \mathbb{Z}_{27}.$

- **Cifrado afín**

$P = C = \mathbb{Z}_{27}$
$K: \{(a, b) \in \mathbb{Z}_{27} \times \mathbb{Z}_{27} : mcd(a, 27) = 1\}$
$Es\ decir, \forall a \in \mathbb{Z}_{27}\ \in \exists a^{-1} \in \mathbb{Z}_{27} : a * a^{-1} \equiv 1\ mod\ 27$
$Para\ k = (a, b) \in K\ definimos$
$e_k(x) = a * x + b\ mod\ 27$
$d_k(y) = a^{-1}(y - b)\ mod\ 27$
$donde\ x\ e\ y\ \in\ \mathbb{Z}_{27}$

- **Cifrado Vigenere**

$Sea\ n \in \mathbb{N}$
$P = C = K = (\mathbb{Z}_{27})^n$
$Para\ k = (k_1, k_2, \dots, k_n)\ \in K, definimos$
$e_k(x_1, x_2, \dots, x_n) = (x_1 + k_1, x_2 + k_2, \dots, x_n + k_n)$
$d_k(y_1, y_2, \dots, y_n) = (y_1 - k_1, y_2 - k_2, \dots, y_n - k_n)$
$donde\ todas\ las\ operaciones\ se\ encuentran\ en\ \mathbb{Z}_{27}.$

- **Cifrado Hill**

$Sea\ n \in \mathbb{N}$
$P = C = (\mathbb{Z}_{27})^n$
$K = \{matrices\ n \times n\ invertibles\ sobre\ \mathbb{Z}_{27}\}$
$Sea\ A \in K, definimos.$
$e_A(x) = xA.$
$d_A(y) = yA^{-1}.$
$donde\ todas\ las\ operaciones\ se\ encuentran\ en\ \mathbb{Z}_{27}, x\ e\ y\ \in (\mathbb{Z}_{27})^n.$

Definición: Cifrado de flujo

Un cifrado de flujo en una 7-tupla (P, C, K, K, F, E, D) que cumple las siguientes condiciones:

1. P es un conjunto finito de posibles textos planos.
2. C es un conjunto finito de posibles textos cifrados.
3. K es un conjunto finito de posibles claves.
4. L es un conjunto finito llamado alfabeto del flujo de claves.
5. $F = (f_1, f_2, f_3, \dots)$ es el generador de flujo de claves. Para $i \geq 1, f_i : K \times P^{i-1} \to L.$
6. Para cada $z \epsilon L$ hay una regla de encriptación $e_z \epsilon E$ y una regla de desencriptación $d_z \epsilon D.$
7. Cada $e_z : P \to C$ y $d_z : P \to C$ son dos funciones que cumplen $d_z(e_z(x)) = x\ \forall x \in P.$

Ejemplo: Cifrado autokey

$P = C = K = L = \mathbb{Z}_{27}$
$Sea\ k\ \in K$
$Sea\ z_1 = k,\ z_i = x_{i-1}\ con\ i \geq 2$
$Para\ cada\ z\ \in\ \mathbb{Z}_{27}\ se\ define$
$e_z(x) = x + z\ mod\ 27$
$d_z(y) = y - z\ mod\ 27$
$donde\ x\ e\ y\ \in \mathbb{Z}_{27}$

E Grupo algebraico

Un grupo es una estructura algebraica formada por un conjunto (G) no vacío dotado de una operación interna ($\circledast$) que combina cualquier par de elementos de G para formar un tercero, dentro del mismo conjunto y satisfaciendo las propiedades: asociativa, existencia del elemento neutro y simétrico [9]. Podemos darle una definición matemática de la siguiente forma:

Definición: Grupo

Sea (G, $\circledast$) un grupo formado por en conjunto no vacío G y la operación binaria interna $\circledast$. Entonces se satisfacen los siguientes axiomas:

1. Asociatividad: $\forall\, a, b, c \,\epsilon\, G : (a \circledast b) \circledast c = a \circledast (b \circledast c)$.
2. Elemento neutro: $\exists e \,\epsilon\, G : a \circledast e = e \circledast a = a$.
3. Elemento simétrico: $\forall a \in G\ \exists a^{-1} \in G :\ a \circledast a^{-1} = a^{-1} \circledast a = e.$

Se dice que un grupo es abeliano si además de cumplir los axiomas anteriores satisface la propiedad conmutativa ($\forall\, a, b \,\epsilon\, G : a \circledast b = b \circledast a$).

Esta estructura algebraica será clave para el correcto funcionamiento de los criptosistemas explicados en el anexo D. Además, sirven como pilar de otras estructuras algebraicas como los cuerpos, anillos o espacios vectoriales.

F Cuerpo

Definición: Cuerpo

Un cuerpo es una estructura algebraica formada por un conjunto no vacío (K) y dos operaciones matemáticas (adición y multiplicación), las cuales satisfacen las propiedades de los grupos abelianos explicadas en el anexo anterior en ambas operaciones matemáticas con una excepción. En el caso del producto, existe un elemento de K que no contiene elemento simétrico, que es el 0 (el cual debe pertenecer al conjunto por la existencia del elemento neutro en la suma).
Además de esas propiedades, también se satisface la propiedad distributiva de la multiplicación respecto a la adición:
$\forall a, b, c \in K, a * (b + c) = (a * b) + (a * c)$.

Definición: Cuerpo finito
Un cuerpo finito es un cuerpo $(K, +, \times)$ cuyo número de elementos incluidos en el conjunto K es un número finito.

Ejemplo:
$\mathbb{Z}_p[x]$ **:** Conjunto de todos los polinomios con indeterminada x y coeficientes en $\mathbb{Z}_p$.
$\mathbb{Z}_p[x] = \{a_n x^n + a_{n-1}x^{n-1} + \cdots + a_1 x + a_0 :\ a_i \in \mathbb{Z}_p,\ n \geq 0\}.$

G AES

Avanced Encription Standard (AES) es un método de cifrado por bloques simétrico publicado por el Instituto Nacional de Normas y Tecnología (del inglés: National Institute of Standards and Technology, NIST) en 2001.

Este estándar surgió como necesidad de solucionar problemas que tenía el estándar anterior, el DES (del inglés: Data Encription Standard). DES había sido roto por fuerza bruta y la solución del triple DES era demasiado lenta. Además, el tamaño de bloque de DES era demasiado pequeño (únicamente 64 bits).

Por estas razones principalmente, el NIST organizó en 1997 un concurso para buscar su sustituto. Fueron eligiendo criterios (como, por ejemplo: coste computacional, memoria necesaria, etc.) hasta que se eligió como ganador el algoritmo llamado Rijndael, desarrollado por dos criptólogos belgas, Joan Daemel y Vicent Rijnmen. Este algoritmo es el que hoy en día conocemos como AES.

Descripción del cifrado

Estrictamente hablando, Rijndael no es exactamente AES ya que hay diferencia entre el tamaño de bloques y longitud de claves. Pero la base matemática que se encuentra tras el algoritmo para aportar la seguridad sí es la misma.

AES utiliza las propiedades de los cuerpos finitos (también llamados campos de Galois) y realiza la mayoría de sus operaciones en un cuerpo finito determinado (más información sobre la parte matemática en el anexo F).

AES utiliza tamaños de claves de 128, 192 o 256 bits y tamaño de bloque fijo de 128 bits. AES opera con una matriz de 4x4 bytes llamada *state*, y realiza cuatro tipos de operaciones distintas sobre esa matriz en un proceso de varias rondas.
Las operaciones que realiza son:

- SubBytes (Figura 0-6): Cada byte del *state* es reemplazado con su entrada en una tabla de búsqueda fija (S). $b_{ij}=S(a_{ij})$.

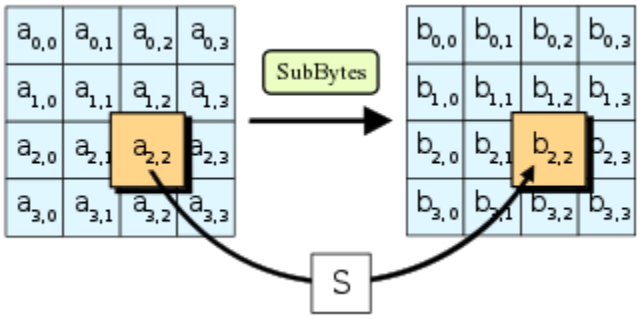

Figura 0-6: Fase SubBytes[5]

[5] Fuente: Wikipedia. Autor: Matt Crypto

- SwitfRows (Figura 0-7): Los bytes en cada fila del *state* rotan de manera cíclica hacia la izquierda, rotando de manera diferente según la fila en la que se encuentran.

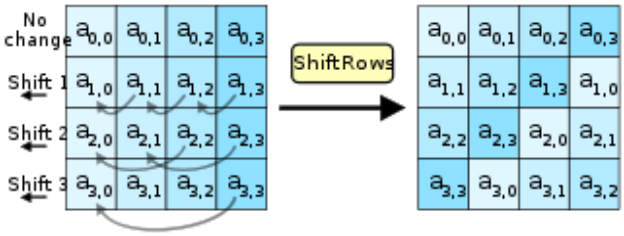

Figura 0-7: Fase ShiftRows[6]

- MixColumns (Figura 0-8): Se multiplican las columnas del *state* por un polinomio constate denominado c(x).

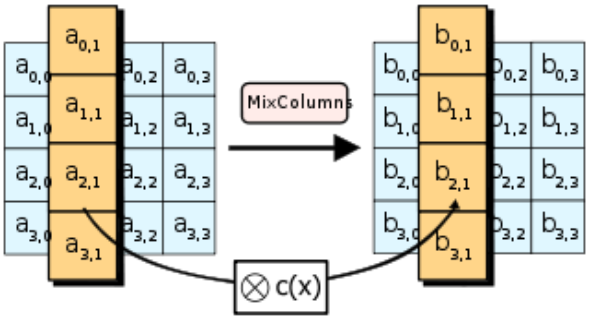

Figura 0-8: Fase MixColumns[7]

- AddRoundKey (Figura 0-9): Cada byte del *state* se combina con un byte de la subclave usando la operación XOR.

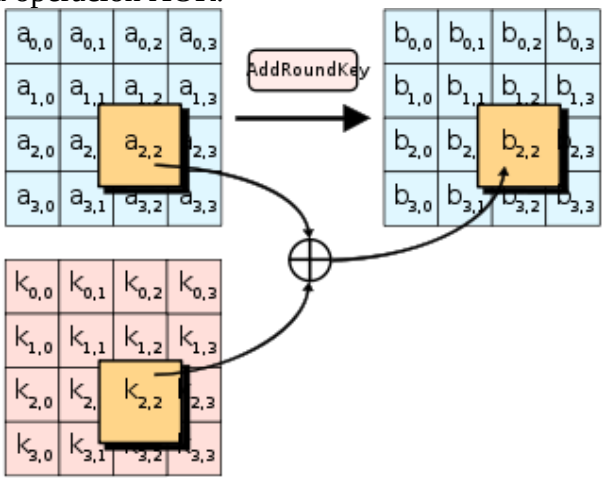

Figura 0-9: Fase AddRoundKey[8]

Bajo estas operaciones hay una gran cantidad de matemáticas que en este trabajo no vamos a explicar. Para resumir, podemos ver que estas operaciones realizan operaciones de

[6] Fuente: Wikipedia. Autor: Matt Crypto.
[7] Fuente: Wikipedia. Autor: Matt Crypto.
[8] Fuente: Wikipedia. Autor: Matt Crypto.

permutación y sustitución parecidas a las explicadas en los criptosistemas del anexo D. Así se consigue llegar a un nivel de cifrado seguro con el que los atacantes no puedan descifrar los datos sin llegar a tener la clave de cifrado.

H RSA

RSA es un algoritmo de criptografía de clave asimétrica. Su nombre se debe a sus desarrolladores (Ronald **R**ivest, Adi **S**hamir y Leonard **A**dleman). Es un algoritmo usado tanto para encriptar como para firmar mensajes. Su seguridad reside en la dificultad de obtener los dos números primos p y q usados para calcular el producto $N = p \times q$.

Generación de claves RSA

Los algoritmos de clave simétrica poseen dos claves para realizar su funcionamiento, una pública y otra privada. Para generar un par de claves RSA se realizan los siguientes pasos:

1. Se eligen dos primos grandes p y q. Se calcula su producto $N = p \times q$.
2. Se calcula la función φ de Euler de N. Aprovechando las propiedades de dicha función y sabiendo que p y q son dos números primos el resultado de esa función es $\varphi(N) = (p-1) \times (q-1)$.
3. Se escoge un número entero e menor que $\varphi(N)$. Requerimos que ese número sea coprimo a $\varphi(N)$ para así garantizar que tiene inverso en aritmética modular.
4. Se determina d, que es el inverso de e en aritmética modular. Es decir, es el número que satisface $e * d \equiv 1 \ mod\ \varphi(N)$.

Entonces, la **clave pública** es **(N,e)** y la **privada (N,d)**. Con estos números se puede demostrar que $m \equiv m^{e*d} \ mod\ N, \forall m \in \mathbb{Z}_N$ pero en este TFG no entraremos en dicha demostración.

Cabe comentar que la clave privada es el único secreto que no conocen el resto de usuarios, así que, si alguien lograra calcular d, la seguridad del algoritmo caería. Si observamos el momento en el que se genera d, vemos que para ello se utiliza la función de Euler, cuyo resultado está determinado por los primos p y q. Entonces, para romper este método de cifrado lo que habría que hacer es calcular estos primos.

Cifrado y descifrado de mensajes RSA

Para cifrar un mensaje lo único que habría que hacer es la siguiente operación, utilizando la clave pública:

$c \equiv m^e \ mod\ N.$

Para descifrar el mensaje, se necesitaría conocer la clave privada para poder hacer la siguiente operación:

$m \equiv c^d \ mod\ N \ \equiv m^{e*d} \ mod\ N$.

I Diffie-Helman

El protocolo criptográfico Diffie-Helman es un protocolo de establecimiento de claves entre dos entidades a través de un canal no seguro de manera anónima [10].
Su seguridad radica en el problema del logaritmo discreto, el cual se considera complejo y mostraremos en el anexo J.

La idea se basa en que dos interlocutores puedan generar una misma clave sin que una tercera persona, la cual podría tener acceso a la información intercambiada, pueda conocerla.

Procedimiento

Alicia y Benito quieren establecer una contraseña mientras una tercera persona, María está observando su comunicación. Para ello se siguen los siguientes pasos:
Se conoce públicamente un número primo p y un generador $g \in \mathbb{Z}_p^*$. Este generador pertenece al conjunto de los enteros menores de p que son primos relativos de p.
Alicia escoge $a \in \mathbb{Z}_{p-1}$ al azar y calcula $A = g^a \ mod \ p$.
Benito escoge $b \in \mathbb{Z}_{p-1}$ al azar y calcula $B = g^b \ mod \ p$.
Ambos intercambian los valores que acaban de calcular (A y B) y son capaces de formar la misma clave K gracias a las propiedades del grupo $\mathbb{Z}_p^*$.
Entonces ambos pueden calcular K gracias a las propiedades del grupo Z.
Alicia calcula $K = B^a \ mod \ p = (g^b \ mod \ p)^a \ mod \ p = g^{b*a} \ mod \ p = g^{a*b} \ mod \ p$.
Benito calcula $K = A^b \ mod \ p = (g^a \ mod \ p)^b \ mod \ p = g^{a*b} \ mod \ p$.

María no puede calcular K ya que solo conoce g, p, A y B. Para poder calcular esta clave debería resolver el problema del logaritmo discreto el cual es extremadamente difícil, aunque hoy en día sigue siendo esto una conjetura y no está demostrado.

J Logaritmo discreto

El logaritmo discreto de y en base g, donde g e y son elementos del grupo finito G, es la solución x de g^x=y. Se puede denotar matemáticamente de la siguiente forma:
$x = log_g(y) \Leftrightarrow g^x = y$.

K Funciones de una sola dirección

Las funciones de una sola dirección son funciones fáciles de computar, pero difícil de invertir. Es decir, fácil de computar significa que un algoritmo las puede computar a tiempo polinomial, mientras que difícil de invertir significa que no existe un algoritmo probabilístico capaz de encontrar la pre-imagen de la función a partir de su salida a tiempo polinomial.

Un ejemplo de funciones de una sola dirección es el logaritmo discreto, pero su existencia es una conjetura. Su existencia implicaría que N no es NP.

L Curvas elípticas

Las matemáticas de las curvas elípticas generan una variante de la criptografía asimétrica a la que denominamos la Criptografía de Curva Elíptica (del inglés: Elliptic Curve Criptography, ECC). Para ver este tipo de criptografía comenzamos viendo que es una cuerva elíptica.

Definición: Curva Elíptica

Una curva elíptica es una curva plana (es decir, una curva que reside en un plano, pudiendo ser abierta o cerrada) que tiene como ecuación la siguiente fórmula:
$y^2 = x^3 + ax + b$

Si llamamos G al conjunto de puntos que forman esa curva (más un punto de infinito) y utilizamos la operación suma tenemos el grupo abeliano (G, +) ya que cumple las propiedades del anexo E. El problema del logaritmo discreto del anexo J sobre este grupo abeliano se cree que es más difícil de resolver que sobre cuerpos finitos. Esta es la razón por la que se cree que las claves en esta criptografía pueden ser más cortas y, aun así, manteniendo un nivel de seguridad similar al de RSA.

Ahora vamos a explicar matemáticamente el funcionamiento de la ECC.

Sea $p > 3$ primo.
Consideramos la curva elíptica $y^2 = x^3 + ax + b$.
Vamos a tratar de demostrar que el conjunto G de puntos que forman esa curva forma un grupo abeliano con una operación aditiva que definiremos más adelante.
En efecto, el conjunto G es el conjunto de soluciones $(x, y)\ \epsilon\ \mathbb{Z}_p \times \mathbb{Z}_p$ que se encuentran en la congruencia $y^2 = x^3 + ax + b \ \ mod\ p$.
Para definir una operación aditiva utilizamos dos puntos de G ($P = (x_1, y_1)$ y $Q = (x_2, y_2)$) y un punto en el infinito (O).
Definimos la operación aditiva de tal forma que si $x_1 = x_2$ e $y_1 = -y_2$, entonces $P + Q = O$.
En el caso contrario tendríamos $P + Q = (x_3, y_3)$, donde x_3 e y_3 siguen las siguientes ecuaciones dependiendo de λ.
$x_3 = \lambda^2 - x_1 - x_2$.
$y_3 = \lambda(x_1 - x_3) - y_1$.
Para finalizar, solo falta definir λ de la siguiente forma:
$\lambda = \frac{y_2 - y_1}{x_2 - x_1}\ si\ P \neq Q$.
$\lambda = \frac{3x_1 + a}{2y_1}\ si\ P = Q$.

De acuerdo a esta definición, conseguimos las propiedades asociativa, conmutativa y de existencia de elemento inverso que mostramos en el anexo E. Para poder concluir que G es un grupo abeliano nos falta definir la propiedad de elemento neutro, la cual definimos de la siguiente forma:

$P + O = O + P = P\ \ \forall P \in G$.

M SHA-256

La función hash SHA-256 pertenece a un conjunto de funciones hash criptográficas denominadas SHA-2. Estas funciones han sido diseñadas por la agencia de seguridad nacional (del inglés: National Security Agency, NSA) como un Estándar Federal de Procesamiento de la información (del inglés: Federal Information Processing Estandard) en 2001.

En un algoritmo que transforma un conjunto arbitrario de elementos de datos en un único valor de longitud fija, el cual se suele utilizar para la verificación de la integridad de un dato original sin tener que proveer del dato original. Podríamos decir que es como una función que devuelve el resumen de una entrada, por eso también se la denomina función resumen.

Su seguridad reside en la capacidad de producir un valor único para cada conjunto de datos de entrada. Si se produce una misma salida para dos entradas distintas, decimos que se ha producido una colisión.

En la Figura 0-10, mostramos un esquema de la fórmula de una iteración en SHA-256:

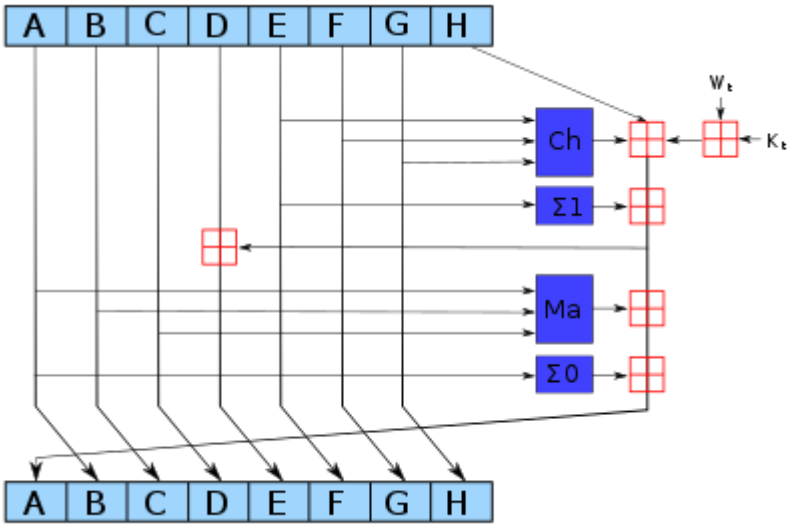

Figura 0-10: Iteración en SHA-256[9]

Las distintas operaciones que se muestran indicadas en la figura anterior son las siguientes:
$Ch(E,F,G) = (E \wedge F) \oplus (\neg E \wedge G)$
$Ma(A,B,C) = (A \wedge B) \oplus (A \wedge C) \oplus (B \wedge C)$
$\Sigma_0(A) = (A \ggg 2) \oplus (A \ggg 13) \oplus (A \ggg 22)$
$\Sigma_1(E) = (E \ggg 2) \oplus (E \ggg 11) \oplus (A \ggg 25)$

[9] Fuente: Wikipedia. Autor: Kockmeyer.

Donde ∧ es un AND lógico, ¬ es un NOR lógico, ⊕ es la función XOR y ⋙ es un desplazamiento de bits.
Además, el cuadrado rojo de la imagen indica una suma con módulo 2^{32}.

N Pruebas de conocimiento cero

Las pruebas de conocimiento cero (del inglés: Zero Knowledge Proof, ZKP) son una serie de protocolos usados por un usuario para demostrar que posee la solución de un problema, mostrando únicamente la veracidad del conocimiento de esta solución [2].

Para explicar este concepto suponemos que Alicia y Benito se encuentran en la cueva de la Figura 0-11. En un determinado punto hay una bifurcación de dos caminos (A y B) conectados más adelante únicamente por una puerta que se encuentra cerrada. Alicia quiere demostrar a Benito que tiene la llave de esa puerta, pero sin tener que enseñarla.

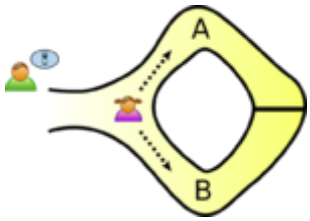

Figura 0-11:Cueva de las ZKP[10]

Para poder hacer eso, Benito espera fuera de la cueva y Alicia toma uno de los dos caminos al azar, el A o el B. Después, entraría Benito a la cueva y le diría a Alicia que se reuniera con él a través de uno de los caminos, el que él eligiera en ese momento.

Si Alicia tuviera la llave, da igual por el camino que hubiera entrado, siempre va a ser capaz de salir por el camino que indique Benito. Mientras que, si no la tiene, no podrá cambiar de camino por el que entró y debería salir siempre por dicho camino, luego tendría un 50% de posibilidades de acertar el camino que indica Bob.

Sin embargo, si Alicia y Benito repiten este experimento muchas veces, la probabilidad de que Alicia no tuviera la llave y acertara todas las peticiones de Benito se reduce a un número muy pequeño, por lo que Alicia terminaría demostrando que en realidad sí que posee esta llave que abre la puerta intermedia.

O Firmas grupales

Con la motivación de abordar esquemas de firmas digitales complejas, se incluye esta sección en este trabajo en la que se comentan unos ejemplos de firmas grupales revocables.

En particular, vamos a comentar brevemente los esquemas KTY04 [19], BSS04 [20] y CPY06.

[10] Fuente: Wikipedia. Autor: Dake.

- **KTY04**

KTY04 es son unos esquemas de firmas grupales revocables que hacen uso de las pruebas de conocimiento cero (anexo N) bajo ciertas suposiciones criptográficas.

La parte de la revocación de estas firmas la hace una entidad "central" aunque esta puede no ser la misma que ha generado el grupo.
Analizando la eficiencia de este tipo de firmas podemos decir que son bastantes costosas computacionalmente, produciendo firmas de un tamaño considerablemente grande (unos 1200 bytes frente a firmas RSA de 256 bytes cuando se usan con tamaño de clave de 2048 bits). Esto ha llevado a producir esquemas de firmas grupales con firmas más cortas como los dos ejemplos que mostramos a continuación.

- **BSS04**

BSS04 es un esquema de firmas grupales revocables basado en las pruebas de conocimiento cero sobre el problema q-SDH.

Este esquema requiere de un administrador de grupo, el cual será capaz de generar las condiciones iniciales del grupo. También se requiere un conjunto de miembros del grupo, los cuales podrán firmar sus mensajes usando sus claves.

A diferencia con el esquema anterior, BSS04 produce firmas de unos 362 bytes con lo que se aprecia un mayor parecido al tamaño de las firmas RSA.

- **CPY06**

CPY06 es un esquema de firmas grupales revocables muy parecido a BSS04 ya que ambos están basados en las pruebas de conocimiento cero sobre el problema q-SDH.

La diferencia entre ambas firmas es que CPY06 no posee la funcionalidad de que un miembro del grupo pueda informar de que una determinada firma ha sido producida por él. Pero esto no afecta al correcto funcionamiento de las firmas grupales.

Además, este esquema produce firmas de tamaño incluyo menor que BSS04, ya que el tamaño de estas firmas suele ser de 192 bytes.

P Autoridad Certificadora

Una autoridad certificadora (del inglés: Certification Authority, CA) es una entidad encargada de firmar certificados digitales. Esta entidad tiene que ser confiable para los usuarios.
Los certificados digitales son documentos que recogen ciertos datos del titular (por ejemplo: su información, su clave pública …) y son firmados por la CA usando su clave privada. La CA se encarga de comprobar que los datos incluidos por el cliente son correctos para así conseguir una verificación de estos.
Las CA se organizan de manera jerárquica, habiendo diferentes niveles entre ellas. Para que las CA puedan ser confiables, también requieren que su certificado se encuentre firmado por una CA de rango superior o incluso por ellos mismos si esto último no es posible. Así se

consigue obtener la confianza en una CA basándose en la confianza que muestra el certificado de la CA raíz de la jerarquía.

Un estándar para infraestructuras de claves públicas es el X.509. X.509 asume un sistema jerárquico estricto de CA. A continuación, mostramos un ejemplo antiguo de un certificado antiguo de www.freesoft.org. En él se puede observar la información del cliente y la firma de una CA para garantizar que estos datos son correctos.

```
Certificate:
   Data:
       Version: 1 (0x0)
       Serial Number: 7829 (0x1e95)
       Signature Algorithm: md5WithRSAEncryption
       Issuer: C=ZA, ST=Western Cape, L=Cape Town, O=Thawte Consulting cc,
               OU=Certification Services Division,
               CN=Thawte Server CA/Email=server-certs@thawte.com
       Validity
           Not Before: Jul  9 16:04:02 1998 GMT
           Not After : Jul  9 16:04:02 1999 GMT
       Subject: C=US, ST=Maryland, L=Pasadena, O=Brent Baccala,
                OU=FreeSoft, CN=www.freesoft.org/Email=baccala@freesoft.org
       Subject Public Key Info:
           Public Key Algorithm: rsaEncryption
           RSA Public Key: (1024 bit)
               Modulus (1024 bit):
                   00:b4:31:98:0a:c4:bc:62:c1:88:aa:dc:b0:c8:bb:
                   33:35:19:d5:0c:64:b9:3d:41:b2:96:fc:f3:31:e1:
                   66:36:d0:8e:56:12:44:ba:75:eb:e8:1c:9c:5b:66:
                   70:33:52:14:c9:ec:4f:91:51:70:39:de:53:85:17:
                   16:94:6e:ee:f4:d5:6f:d5:ca:b3:47:5e:1b:0c:7b:
                   c5:cc:2b:6b:c1:90:c3:16:31:0d:bf:7a:c7:47:77:
                   8f:a0:21:c7:4c:d0:16:65:00:c1:0f:d7:b8:80:e3:
                   d2:75:6b:c1:ea:9e:5c:5c:ea:7d:c1:a1:10:bc:b8:
                   e8:35:1c:9e:27:52:7e:41:8f
               Exponent: 65537 (0x10001)
   Signature Algorithm: md5WithRSAEncryption
       93:5f:8f:5f:c5:af:bf:0a:ab:a5:6d:fb:24:5f:b6:59:5d:9d:
       92:2e:4a:1b:8b:ac:7d:99:17:5d:cd:19:f6:ad:ef:63:2f:92:
       ab:2f:4b:cf:0a:13:90:ee:2c:0e:43:03:be:f6:ea:8e:9c:67:
       d0:a2:40:03:f7:ef:6a:15:09:79:a9:46:ed:b7:16:1b:41:72:
       0d:19:aa:ad:dd:9a:df:ab:97:50:65:f5:5e:85:a6:ef:19:d1:
       5a:de:9d:ea:63:cd:cb:cc:6d:5d:01:85:b5:6d:c8:f3:d9:f7:
       8f:0e:fc:ba:1f:34:e9:96:6e:6c:cf:f2:ef:9b:bf:de:b5:22:
       68:9f
```

Figura 0-12: Ejemplo certificado X.509

Q Red TOR

La red TOR es una red con enrutamiento de cebolla en la que una serie de nodos se comunican con el protocolo TLS sobre TCP/IP. Podemos dividir las entidades que forman esta estructura en dos:

- Nodos OR (del inglés: Onion Router, OR): Son los nodos intermedios del sistema. Funcionan como encaminadores, y forman entre ellos la ruta que seguirá la información hasta su destino. Se denomina "exit node" al último nodo del servicio. El primer nodo se denomina "entry node" y los otros nodos, "middle node".
- Nodos OP (del inglés: Onion Proxy, OP): Su funcionalidad es establecer el circuito aleatorio que seguirá la información entre el nodo origen y el destino. También manejará las conexiones de aplicaciones del usuario.

Para explicar su funcionamiento podemos hacer referencia a la siguiente figura:

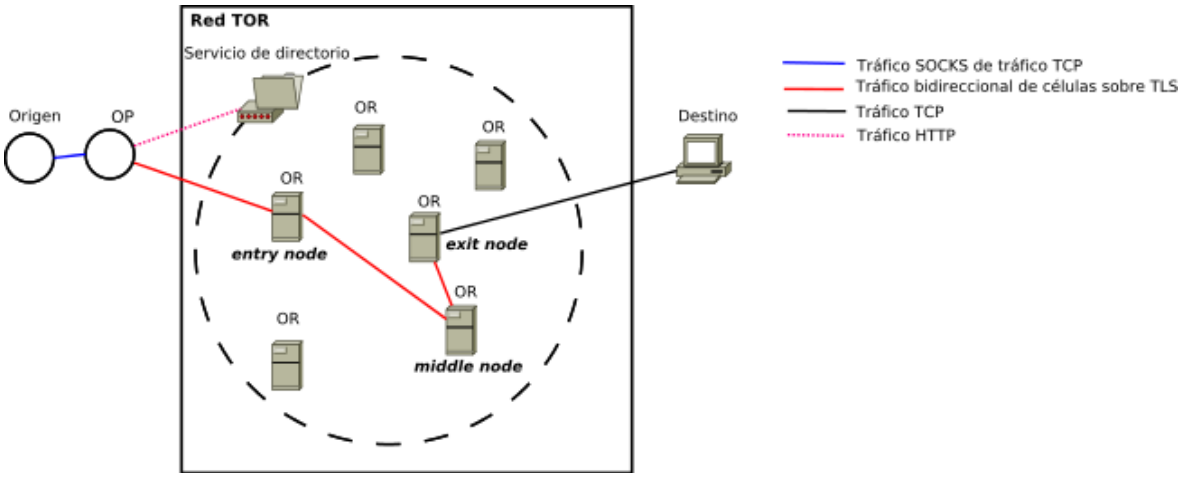

Figura 0-13: Esquema red TOR[11]

- OP determina el circuito de 3 nodos OR que seguirán los paquetes. Se apoya en el servidor de directorio y en la información incluida en su configuración.
- OP negocia con los nodos OR las claves necesarias para el funcionamiento del circuito antes de realizar ninguna transmisión previa. El intercambio de claves se produce usando Diffie-Helman del anexo I para así llegar a obtener claves simétricas AES compartidas.
- Los mensajes irán cifrados usando varias capas de cifrado cuyas claves son las claves simétricas que previamente ha negociado OP. De esta forma tenemos un enrutamiento de cebolla como en la sección 2.5.2.
- Cada nodo OP va "pelando" la capa externa del cifrado, ya que no contiene información suficiente para descifrar el resto. La información que encuentra en este proceso es únicamente la siguiente dirección que el paquete debe seguir. De esta forma, los nodos OP no conocen ni la ruta en su totalidad, ni el mensaje que se está transmitiendo.
- El último nodo OP ya sí que puede conocer la dirección final del paquete, así que así se consigue llegar al destino de una manera correcta.

[11] Fuente: Wikipedia. Autor: Fercufer.

R Aplicación Android

En esta sección explicaremos brevemente el trabajo que realizan las actividades de la aplicación Android para poder seguir con el diseño del protocolo del capítulo 3.

La aplicación comienza con una actividad llamada Splash_Activity, la cual muestra una primera imagen de nuestra aplicación mientras esta se está cargando, como se puede observar en la Figura 0-14:

Figura 0-14: Splash Activity

Después, si no hay una sesión activa, se pasa automáticamente a la actividad llamada Login_Activity. En esta actividad se pueden introducir los datos del cliente para que este pueda empezar la Fase 1 del protocolo, explicada en la sección 3.3.1. Antes de ello, el cliente tiene que leer y aceptar un contrato, como podemos ver en la Figura 0-15. Una vez estén introducidos los datos del cliente y firmado el contrato como se muestra en la Figura 0-16, la actividad iniciará la Fase 1, llamando a la función específica que lo ejecuta y recibiendo de ella tres variables (ID, T y Val_M) las cuales guardará como parte de la sesión activa.

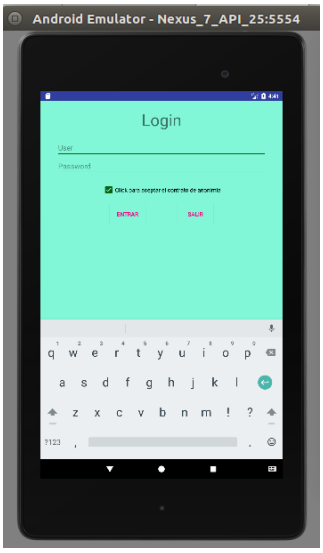

Figura 0-15: Login Activity

Figura 0-16: Contract Activity

Una vez se encuentre la sesión activa (es decir, una vez la aplicación posea las tres variables previamente mencionadas), se lanzará la actividad Main_Activity, en la cual se incluirán las demás fases del protocolo en la parte de crear la actividad de la siguiente forma:

- Comienza llamando a la función que realiza la Fase 2 de la sección 3.3.2. Esta función recibirá las tres variables obtenidas anteriormente (ID, T y Val_M) y devolverá dos nuevas variables que se usarán para calcular la clave del sistema de la sección 3.7 (T_1 y Salt).
- Después, realiza la Fase 3 de la sección 3.3.3. En esta ocasión solo recibirá como respuesta si se ha conectado correctamente (ver Figura 0-17) o si ha habido un error, concluyendo así a parte de creación de la actividad.

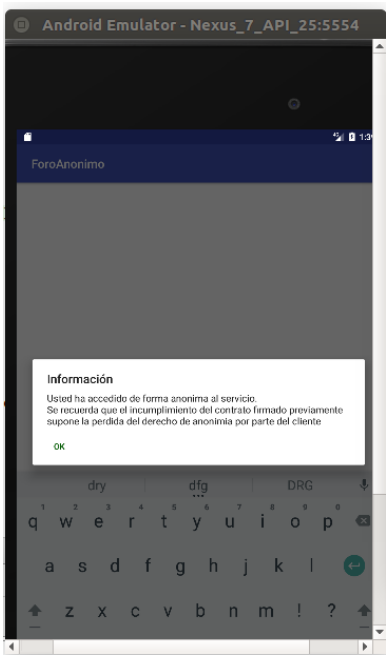

Figura 0-17: Acceso correcto

Para finalizar la Fase 3 o comenzar la Fase 4, es necesario realizar una petición de logout. Esta parte ya no se realiza mientras la actividad se está creando, sino cuando se pide la desconexión al servidor. El cliente volverá a la actividad de Login_Activity tras este paso y recibirá dos mensajes diferentes dependiendo de la fase que se haya ejecutado (la cual depende de las acciones que él mismo ha realizado). Si se ejecuta la Fase 3, el cliente no recibirá ninguna notificación indicando que todo el proceso ha sido ejecutado correctamente. Mientras que, si se ejecuta la Fase 4, el cliente podrá leer el siguiente mensaje de la Figura 0-18 y se le enviará a su correo electrónico información más detallada acerca de su revocación.

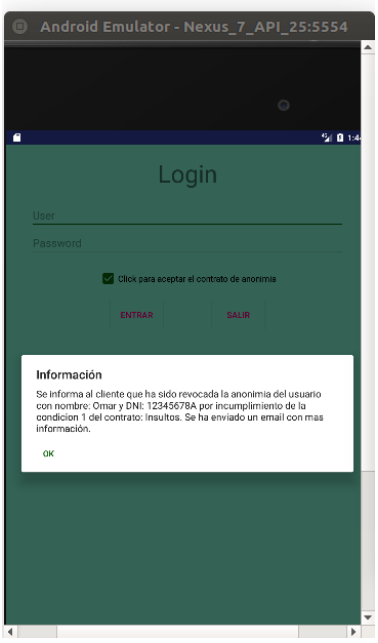

Figura 0-18: Mensaje revocación

S Ataques al sistema

En esta sección introducimos brevemente tres tipos de ataques informáticos que se han analizado en la parte 5.5 del trabajo.

Ataque de intermediario (Man in the middle)

Es un ataque en el que una tercera persona es capaz de observar e interceptar mensajes entre dos entidades, sin que estas lo lleguen a conocer. El atacante podría leer, insertar o modificar mensajes a su voluntad, por lo que se requiere una serie de medidas de seguridad para poder evitar que el atacante pueda hacer daño al sistema.

El principal ejemplo que podríamos comentar es durante el intercambio de claves de Diffie-Helman del anexo I cuando este se realiza sin autentificación. El atacante podría interceptar la conversación y hacerse pasar por una entidad, determinando así una clave compartida con la otra entidad sin que esta llegue a saber que en realidad está intercambiando información que puede descifrar el atacante.

DDOS

Se llama así al ataque de denegación de servicio (del inglés: Denial of Service, DoS). Es un ataque a un sistema de computadoras o red que produce su inhabilitación, haciendo inaccesible a los usuarios legítimos.

SQL Inyection

El ataque SQL Inyection consiste en introducir datos maliciosos a un sistema que opera con bases de datos para conseguir información específica, o realizar cambios en las bases de datos según la voluntad del atacante.

Un ejemplo sencillo es la introducción de cadenas de caracteres especiales en las consultas en las bases de datos. Si el sistema no tiene en cuenta estas posibles amenazas, podría producirse la situación siguiente.

El sistema posee una query sin filtros para obtener la información de un usuario dado su nombre y su contraseña:
*SELECT * FROM Users WHERE User= %s AND Password = Hash(%s);*

Se introduce como nombre y contraseña de usuario los siguientes datos malintencionados del cliente:

User: *proof ; DROP TABLE Users ; #*
Password: *proof2*

Entonces la query que consultaría nuestro sistema sería:
*SELECT * FROM Users WHERE User= proof ; DROP Table Users ; # AND Password = Hash (proof2);*

Y lo que estaríamos haciendo con esta petición sería realizar una consulta no esperada y después, borrar toda la base de datos. (Ya que todo lo que va después de # se considera comentario y no se ejecuta).

T Código

El código lo podemos dividir en dos partes dependiendo del lenguaje de programación empleado: Java y Python.
En este anexo resumimos la parte del código implementada más importante relacionada con el diseño propuesto del capítulo 3, en el que se incluye toda la funcionalidad explicada en el capítulo 4.

Java

En la parte de java se encuentra implementado el actor Ui. Vamos a destacar tres clases que intervienen de manera directa en el protocolo criptográfico de la sección 3.3.

Clase ClientApp.java:
Esta es la clase más importante en la que se encuentra la parte de Ui en todas las fases del protocolo. También contiene métodos para la lectura y escritura de mensajes según la sección 3.5 y sobre el cifrado de estos mensajes explicado en la sección 3.6. Por último, se incluye la función determinista de creación de claves de acceso al sistema de la sección 3.7.

```
1. import android.content.Context;
2. import android.util.Log;
3.
4. import java.io.ByteArrayOutputStream;
5. import java.net.Socket;
6. import java.util.Arrays;
7. import java.util.Date;
8. import java.util.Random;
9.
10.      import es.uam.eps.dadm.damas.model.RoundRepository;
11.
12.      /*Clase ClientApp
13.      * Contiene la funcionalidad requerida por parte del actor Ui
   del sistema para
14.      * poder realizar el protocolo criptografico implementado
15.      * En esta clase tambien se incluyen metodos secundarios
   usados para enviar y recibir mensajes
16.      * dentro del protocolo
17.      * @author: Omar Molinero Gil
18.      * @version: 18/06/18-v13*/
19.
20.      public class ClientApp implements Runnable{
21.
22.          // Contexto de la aplicacion
23.          private Context c;
24.
25.          //Cifrado RSA
26.          private RSACrypto RSA;
27.
28.          //Clave simetrica AES
```

```
29.          private static byte[] key;
30.
31.          //Socket que establecera la conexion entre los actores
32.          private static Socket sk;
33.
34.
35.          /*CONSTANTES SOBRE LA DIRECION IP Y PUERTOS DE LOS
   ACTORES DEL SISTEMA*/
36.          private final String IP="10.0.2.2";
37.          private final int SU_PORT=10000;
38.          private final int M_PORT=9000;
39.          private final int SM_PORT=8000;
40.
41.
42.          /*DECLARACION CONSTANTES CODIGOS MENSAJES*/
43.          private final int P_REGISTRO=1;
44.          private final int P_ACUERDO_M=2;
45.          private final int P_ACUERDO_M_RESPUESTA=3;
46.          private final int P_OK_REGISTRO=4;
47.          private final int P_ACCESO_M=5;
48.          private final int P_ACCESO_M_RESPUESTA=6;
49.          private final int P_ACCESO_M_SALT=7;
50.          private final int P_OK_ACCESO_M=8;
51.          private final int P_ACCESO_SM=9;
52.          private final int P_ACCESO_SM_RESPUESTA=10;
53.          private final int P_ACCESO_SM_TOKEN=11;
54.          private final int P_LOGOUT=12;
55.          private final int P_PETICION_TOKEN_M=13;
56.          private final int P_PETICION_TOKEN_M_RESPUESTA=14;
57.          private final int P_OK_ACCESO_SM=15;
58.          private final int P_PETICION_REVOCACION_M=16;
59.          private final int P_PETICION_REVOCACION_M_RESPUESTA=17;
60.          private final int P_PETICION_REVOCACION_SU=18;
61.          private final int P_PETICION_REVOCACION_SU_RESPUESTA=19;
62.          private final int P_OK_REVOCACION=20;
63.          private final int P_ERROR_MSG=-1;
64.          private final int P_ERROR_FIRMA=-2;
65.          private final String SEPARADOR_CODIGO=";";
66.          private final String SEPARADOR_DATOS=":";
67.
68.
69.
70.          /*DECLARACION CONSTANTES CLAVES RSA*/
71.          public static final int SU_PUBLIC=1;
72.          public static final int M_PUBLIC=2;
73.          public static final int SM_PUBLIC=3;
74.
75.
76.          /*DECLARACION CONSTANTES PROTOCOLOS*/
77.          private final int PROTOCOLO_1=1;
78.          private final int PROTOCOLO_2=2;
79.          private final int PROTOCOLO_3=3;
80.          private final int PROTOCOLO_4=4;
81.
82.          /*VARIABLES INICIALIZACION*/
83.          private int protocolo;
84.          private String DNI;
85.          private String password;
86.          private RoundRepository.LoginRegisterCallback callback;
87.          private RoundRepository.AnonymousAccessCallback callback2
   ;
```

```
88.          private String valM;
89.          private String ID;
90.          private String salt;
91.          private String T0;
92.          private String T1;
93.
94.
95.          /*ClientApp(c,DNI,password,callback)
96.           * Constructor del protocolo 1
97.           * @param: c, Contexto de la aplicacion
98.           * @param: DNI, documento de identidad del usuario
99.           * @param: password, contrasenya secreta del usuario para
   poder registrarse con SU
100.          * @param: callback, funcion usada para obtener las
   variables del resultado de ejecutar este protocolo
101.          *          en la actividad que lo llame (login_activity)
102.          * */
103.         public ClientApp(Context c, String DNI, String password,
   RoundRepository.LoginRegisterCallback callback){
104.             this.c=c;
105.             this.DNI=DNI;
106.             this.password=password;
107.             this.callback=callback;
108.             this.protocolo=PROTOCOLO_1;
109.         }
110.
111.
112.         /*ClientApp(c,valM,ID,T0,callback2)
113.          * Constructor del protocolo 2
114.          * @param: c, Contexto de la aplicacion
115.          * @param: valM, token de acceso a M
116.          * @param: ID, identificador numerico del usuario
117.          * @param: T0, marca temporal del usuario
118.          * @param: callback2, funcion usada para obtener las
   variables del resultado de ejecutar este protocolo
119.          *          en la actividad que lo llame (main_activity)
120.          * */
121.         public ClientApp(Context c, String valM, String ID, Strin
   g T0,RoundRepository.AnonymousAccessCallback callback2){
122.             this.c=c;
123.             this.valM=valM;
124.             this.ID=ID;
125.             this.T0=T0;
126.             this.callback2=callback2;
127.             this.protocolo=PROTOCOLO_2;
128.         }
129.
130.         /* ClientApp(c,valM,ID,T0,callback2)
131.          * Constructor del protocolo 3
132.          * @param: c, Contexto de la aplicacion
133.          * @param: T1, marca temporal de solicitud de acceso al
   sistema
134.          * @param: ID, identificador numerico del usuario
135.          * @param: T0, marca temporal del usuario
136.          * @param: salt, valor aleatorio producido por el usuario
   para anyadir entropia
137.          * */
138.         public ClientApp(Context c, String ID, String T0, String
   T1, String salt){
139.             this.c=c;
140.             this.ID=ID;
```

```
141.             this.T0=T0;
142.             this.T1=T1;
143.             this.salt=salt;
144.             this.protocolo=PROTOCOLO_3;
145.         }
146.
147.         /* ClientApp(c,valM,ID,T0,callback2)
148.          * Constructor del protocolo 3*
149.          * @param: c, Contexto de la aplicacion
150.          * */
151.         public ClientApp(Context c){
152.             this.c=c;
153.             this.protocolo=PROTOCOLO_4;
154.         }
155.
156.         /* run()
157.          * Metodo runneable que se ejecutara cuando se quiera
   establecer comunicacion con algun actor
158.          * del sistema. La fase del protocolo que se lanzara
   depende de la inicializacion de la clase
159.          * */
160.         @Override
161.         public void run(){
162.             if (this.protocolo==PROTOCOLO_1)
163.                 P1();
164.             else if(this.protocolo==PROTOCOLO_2)
165.                 P2();
166.             else if(this.protocolo==PROTOCOLO_3)
167.                 P3();
168.             else if(this.protocolo==PROTOCOLO_4)
169.                 P4();
170.         }
171.
172.
173.         /* P1()
174.          * Metodo que ejecuta las acciones de Ui en la primera
   fase del protocolo criptografico
175.          * */
176.         private void P1() {
177.             // Declaramos el metodo de cifrado RSA que usara la
   clave publica de SU para descifrar sus mensajes
178.             // ya que la primera fase del protocolo, este metodo
   gestiona los mensajes enviados entre SU y Ui
179.             // Para mayor informacion, mirar el disenyo del
   protocolo
180.             RSA= new RSACrypto(c,SU_PUBLIC);
181.
182.             try {
183.                 //Declaracion de variables que usaremos para
   recibir los mensajes de SU.
184.                 byte[] bytes = new byte[128];
185.                 byte[] bytes2 = new byte[128];
186.
187.                 //Mensaje de seguimiento
188.                 Log.d("PROTOCOLO 1","EMPEZAMOS LA FASE 1 DEL
   PROTOCOLO");
189.
190.                 //Nos conectamos con SU
191.                 sk = new Socket(IP, SU_PORT);
192.
193.                 //Mensaje de seguimiento
```

```
194.                 Log.d("PROTOCOLO 1", "Conexion establecida con
   SU");
195.
196.
197.                 //Preparamos los datos a enviar en el primer
   mensaje del protocolo
198.                 String[] datos=new String[2];
199.                 datos[0]=DNI;
200.                 datos[1]=password;
201.                 //ENVIAMOS EL PRIMER MENSAJE DE REGISTRO
202.                 Log.d("PROTOCOLO 1", "Enviamos el mensaje de
   registro de la fase 1 del protocolo");
203.                 send_msg(make_msg(P_REGISTRO,datos));
204.
205.
206.                 //LEEMOS LA RESPUESTA DEL SERVIDOR
207.                 sk.getInputStream().read(bytes);
208.                 sk.getInputStream().read(bytes2);
209.
210.
211.                 //Comprobamos que la firma de SU recibida es
   correcta
212.                 if(!RSA.verify(bytes,bytes2)) {
213.                     //mensaje de seguimiento
214.                     Log.d("PROTOCOLO 1", "Hemos recibido una
   firma no valida.");
215.
216.                     //Regresemos a la actuvidad indicando que
   algo ha salido mal
217.                     callback.onError("FIRMA RECIBIDA NO VALIDA
   PROTOCOLO 1");
218.                     return;
219.                 }
220.
221.                 //Si la firma es correcta, desciframos el mensaje
   recibido
222.                 String recibimos= new String(RSA.decrypt(bytes));
223.                 //mensaje de seguimiento
224.                 Log.d("PROTOCOLO 1","Recibimos de SU:
   "+recibimos);
225.
226.                 //Extraemos los datos incluidos en el mensaje
227.                 String[] tokens=read_msg(recibimos);
228.                 if (tokens.length!=3 || get_code_msg(recibimos)!=
   P_OK_REGISTRO) {
229.                     Log.d("PROTOCOLO 1", "No hemos recibido los
   datos esperados en el mensaje");
230.                     callback.onError("ERROR POR RECIBIR DATOS NO
   ESPERADOS");
231.                     return;
232.                 }
233.
234.                 //Mandamos esos datos a la actividad para que los
   gestione.
235.                 //En particular, estos datos son ID, T y ValM
236.                 Log.d("PROTOCOLO 1","FIN DE LA FASE 1 DEL
   PROTOCOLO "+recibimos);
237.                 callback.onLogin(tokens[2],tokens[1],tokens[0]);
238.
239.
240.
```

```
241.             } catch (Exception e) {
242.                 e.printStackTrace();
243.                 Log.d("PROTOCOLO 1", "Ha ocurrido un error
   inesperado" +e.getMessage());
244.                 callback.onError("ERROR. HA SALTADO UNA
   EXCEPCION");
245.
246.             }
247.
248.         }
249.
250.         /* P2()
251.          * Metodo que ejecuta las acciones de Ui en la segunda
   fase del protocolo criptografico
252.          * */
253.         private void P2() {
254.             // Declaramos el metodo de cifrado RSA que usara la
   clave publica de M para descifrar sus mensajes
255.             // ya que la segunda fase del protocolo, este metodo
   gestiona los mensajes enviados entre M y Ui
256.             // Para mayor informacion, mirar el disenyo del
   protocolo
257.             RSA= new RSACrypto(c,M_PUBLIC);
258.
259.             try {
260.                 //Declaracion de una variable que usaremos para
   recibir los mensajes de M.
261.                 byte[] bytes;
262.
263.                 //Mensaje de seguimiento
264.                 Log.d("PROTOCOLO 2","EMPEZAMOS LA FASE 2 DEL
   PROTOCOLO");
265.
266.                 //Nos conectamos con M
267.                 sk = new Socket(IP, M_PORT);
268.
269.                 //Mensaje de seguimiento
270.                 Log.d("PROTOCOLO 1", "Conexion establecida con
   SU");
271.
272.                 //Calculamos una nueva marca temporal y una clave
   AES
273.                 Date t = new Date();
274.                 ClientApp.key=AESCrypto.gernetarekey();
275.
276.
277.                 //Preparamos los datos a enviar en el primer
   mensaje del protocolo
278.                 String[] datos=new String[3];
279.                 datos[0]=valM;
280.                 datos[1]=String.valueOf(t.getTime());
281.                 datos[2]=ID;
282.
283.                 Log.d("PROTOCOLO 2", "Enviamos el mensaje de
   acceso a M de la fase 2 del protocolo");
284.
285.
286.                 //ENVIAMOS EL MENSAJE DE ACCESO, cifrado con RSA
   y firma RSA
287.                 send_msg(make_msg(P_ACCESO_M,datos),ClientApp.key
   );
```

```
288.
289.
290.                 //LEEMOS LA RESPUESTA DEL SERVIDOR, Recordamos
   que nos enviara un mensaje cifrado con AES sin firma
291.                 int a;
292.                 while((a=sk.getInputStream().available())==0){
293.                 }
294.                 bytes=new byte[a];
295.                 sk.getInputStream().read(bytes);
296.
297.
298.                 //Desciframos el mensaje recibido usando AES
299.                 String recibimos= new String(AESCrypto.decrypt(Cl
   ientApp.key, Arrays.copyOfRange(bytes, 0, 16),Arrays.copyOfRange(by
   tes, 16, bytes.length)));
300.                 //Mensaje de seguimiento
301.                 Log.d("PROTOCOLO 2","Recibimos de M:
   "+recibimos);
302.
303.                 //Leemos los datos enviados en el mensaje de M
304.                 String[] tokens=read_msg(recibimos);
305.
306.                 //Creamos una clase generadora de numeros
   aleatorios
307.                 Random r =new Random();
308.                 //Creamos un long aleatorio que sera la variable
   Salt
309.                 long salt=r.nextLong();
310.
311.                 //Mensaje de seguimiento
312.                 Log.d("PROTOCOLO 2","Variable salt creada:
   "+salt);
313.                 datos=new String[2];
314.                 datos[0]=String.valueOf(salt);
315.                 datos[1]=tokens[1];
316.
317.                 //ENVIAMOS EL SIGUIENTE MENSAJE DE ACCESO, ahora
   cifrado con AES
318.                 send_msg_AES(make_msg(P_ACCESO_M_SALT,datos));
319.
320.
321.                 //ESPERAMOS RESPUESTA OK DEL SERVIDOR
322.                 while((a=sk.getInputStream().available())==0){
323.                 }
324.                 bytes=new byte[a];
325.                 sk.getInputStream().read(bytes);
326.
327.                 //Desciframos el mensaje leido
328.                 recibimos= new String(AESCrypto.decrypt(ClientApp
   .key, Arrays.copyOfRange(bytes, 0, 16),Arrays.copyOfRange(bytes, 16
   , bytes.length)));
329.                 Log.d("PROTOCOLO 2","Recibimos de M "+recibimos);
330.
331.                 //Si recibimos un mensaje indicando que la fase 2
   del protocolo se ha realizado correctamente
332.                 if(get_code_msg(recibimos)==P_OK_ACCESO_M){
333.                     Log.d("PROTOCOLO 2","FIN DE LA FASE 2 DEL
   PROTOCOLO");
334.                     //Mandamos esos datos a la actividad para que
   los gestione.
335.                     //En particular, estos datos son salt y T1
```

```
336.                     this.callback2.onAccess(tokens[1],salt);
337.                 }else{
338.                     Log.d("PROTOCOLO 2","Error en la fase 2 del
   protocolo");
339.                     this.callback2.onError("RECIBIMOS MENSAJE NO
   ESPERADO");
340.                 }
341.
342.
343.
344.
345.             } catch (Exception e) {
346.                 e.printStackTrace();
347.                 Log.d("PROTOCOLO 2", "Ha ocurrido un error
   inesperado" +e.getMessage());
348.                 callback.onError("ERROR. HA SALTADO UNA
   EXCEPCION");
349.             }
350.
351.         }
352.
353.         /* P3()
354.          * Metodo que ejecuta las acciones de Ui en la tercera
   fase del protocolo criptografico
355.          * */
356.         private void P3() {
357.             // Declaramos el metodo de cifrado RSA que usara la
   clave publica de SM para descifrar sus mensajes
358.             // ya que la tercera fase del protocolo, este metodo
   gestiona los mensajes enviados entre SM y Ui
359.             // Para mayor informacion, mirar el disenyo del
   protocolo
360.             RSA= new RSACrypto(c,SM_PUBLIC);
361.
362.             try {
363.                 //Declaracion de variables que usaremos para
   recibir los mensajes de SM.
364.                 byte[] bytes;
365.
366.                 //Mensaje de seguimiento
367.                 Log.d("PROTOCOLO 3","EMPEZAMOS LA FASE 3 DEL
   PROTOCOLO");
368.
369.                 //Nos conectamos con SM
370.                 ClientApp.sk = new Socket(IP, SM_PORT);
371.
372.                 //Calulamos una nueva marca temporal y una nueva
   clave AES
373.                 Date t = new Date();
374.                 ClientApp.key=AESCrypto.gernetarekey();
375.
376.                 //Preparamos los datos a enviar en el primer
   mensaje del protocolo
377.                 String[] datos=new String[1];
378.                 datos[0]=String.valueOf(t.getTime());
379.
380.                 Log.d("PROTOCOLO 2", "Enviamos el mensaje de
   acceso a SM de la fase 3 del protocolo");
381.                 //ENVIAMOS EL MENSAJE DE ACCESO, cifrado con RSA
   y firma RSA
```

```
382.                 send_msg(make_msg(P_ACCESO_SM,datos),ClientApp.ke
   y);
383.
384.
385.                 //LEEMOS LA RESPUESTA DEL SERVIDOR, Recordamos
   que nos enviara un mensaje cifrado con AES sin firma
386.                 int a;
387.                 while((a=sk.getInputStream().available())==0){
388.                 }
389.                 bytes=new byte[a];
390.                 sk.getInputStream().read(bytes);
391.
392.                 //Desciframos el mensaje recibido usando AES
393.                 String recibimos= new String(AESCrypto.decrypt(Cl
   ientApp.key, Arrays.copyOfRange(bytes, 0, 16),Arrays.copyOfRange(by
   tes, 16, bytes.length)));
394.                 //Mensaje de seguimiento
395.                 Log.d("PROTOCOLO 3","Recibimos de SM:
   "+recibimos);
396.
397.                 String[] tokens=read_msg(recibimos);
398.
399.                 //Enviamos el token de acceso al servicio
400.                 //send_msg_AES(make_msg_access_SM2(tokens[1]),act
   _SaltyHash());
401.
402.
403.                 sk.getOutputStream().write(act_SaltyHash());
404.
405.                 //LEEMOS LA RESPUESTA DEL SERVIDOR, Recordamos
   que nos enviara un mensaje cifrado con AES sin firma
406.                 while((a=sk.getInputStream().available())==0){
407.                 }
408.                 bytes=new byte[a];
409.                 sk.getInputStream().read(bytes);
410.
411.                 //Desciframos el mensaje recibido usando AES
412.                 recibimos= new String(AESCrypto.decrypt(ClientApp
   .key, Arrays.copyOfRange(bytes, 0, 16),Arrays.copyOfRange(bytes, 16
   , bytes.length)));
413.                 Log.d("PROTOCOLO 3","Recibimos de SM
   "+recibimos);
414.
415.                 //El codigo del mensaje que recibimos decidira si
   hemos accedido correctamente o no
416.                 if(get_code_msg(recibimos)==P_OK_ACCESO_SM){
417.                     Log.d("PROTOCOLO 3","HEMOS ACCEDIDO
   CORRECTAMENTE AL SERVICIO ");
418.                 }else{
419.                     Log.d("PROTOCOLO 3","NO HEMOS ACCEDIDO
   CORRECTAMENTE ");
420.                 }
421.
422.
423.             } catch (Exception e) {
424.                 e.printStackTrace();
425.                 Log.d("PROTOCOLO 3", "Ha ocurrido un error
   inesperado" +e.getMessage());
426.                 callback.onError("ERROR. HA SALTADO UNA
   EXCEPCION");
427.             }
```

```
428.
429.         }
430.
431.         /* P4()
432.          * Metodo que ejecuta las acciones de Ui en la tercera o
   cuarta fase del protocolo criptografico
433.          * */
434.         private void P4() {
435.             //El comienzo de este metodo es el envio de la
   peticion de desconexion al servidor, luego
436.             // este metodo ejecuta el final de la tercera fase o
   el principio de la cuarta fase segun el disenyo
437.             // Para ello hay que tener en cuenta las variables ya
   guardadas ClientApp.sk y ClientApp.key
438.
439.             //Mensaje de seguimiento
440.             Log.d("PROTOCOLO 3/4","PETICION DE DESCONEXION AL
   SISTEMA");
441.
442.             try {
443.                 //Declaracion de variables que usaremos para
   recibir los mensajes de SM.
444.                 byte[] bytes;
445.
446.                 //Preparamos los datos a enviar en el primer
   mensaje del protocolo
447.                 String[] datos=new String[1];
448.                 datos[0]="Peticion de logout";
449.
450.                 Log.d("PROTOCOLO 3/4", "Enviamos el mensaje de
   peticion de logout a SM de la fase 3/4 del protocolo");
451.                 //ENVIAMOS EL MENSAJE DE LOGOUT, cifrado con AES
452.                 send_msg_AES(make_msg(P_LOGOUT,datos));
453.
454.                 //LEEMOS LA RESPUESTA DEL SERVIDOR, Recordamos
   que nos enviara un mensaje cifrado con AES sin firma
455.                 int a;
456.                 while((a=sk.getInputStream().available())==0){
457.                 }
458.                 bytes=new byte[a];
459.                 sk.getInputStream().read(bytes);
460.
461.                 //Desciframos el mensaje recibido usando AES
462.                 String recibimos= new String(AESCrypto.decrypt(Cl
   ientApp.key, Arrays.copyOfRange(bytes, 0, 16),Arrays.copyOfRange(by
   tes, 16, bytes.length)));
463.                 //Mensaje de seguimiento
464.                 Log.d("PROTOCOLO 3/4","Recibimos de SM:
   "+recibimos);
465.
466.                 //Recibimos los datos incluidos en el mensaje
   enviado
467.                 String[] tokens=read_msg(recibimos);
468.
469.                 //El codigo del mensaje que recibimos decidira si
   estamos en la fase 3 o fase 4
470.                 if(get_code_msg(recibimos)==P_OK_ACCESO_SM){
471.                     Log.d("PROTOCOLO 3","NOS HEMOS DESCONECTADO
   CORRECTAMENTE DEL SERVICIO ");
472.                 }else{
```

```
473.                     Log.d("PROTOCOLO 4","NUESTRA IDENTIDAD HA
   SIDO REVOCADA");
474.                 }
475.
476.             } catch (Exception e) {
477.                 e.printStackTrace();
478.                 Log.d("PROTOCOLO 3/4", "Ha ocurrido un error
   inesperado" +e.getMessage());
479.                 callback.onError("ERROR. HA SALTADO UNA
   EXCEPCION");
480.             }
481.
482.         }
483.
484.         /* send_msg(msg)
485.          * Metodo interno usado para enviar mensajes en el
   protocolo cifrados y firmados con RSA
486.          * @param: msg, Mensaje sin cifrar siguiendo la
   estructura de mensajes del protocolo
487.          * */
488.         private void send_msg(String msg) throws Exception {
489.             byte[] enviamos=RSA.encrypt(msg.getBytes());
490.             byte[] firmado=RSA.firma(enviamos);
491.             ByteArrayOutputStream outputStream = new ByteArrayOut
   putStream();
492.             outputStream.write(enviamos);
493.             outputStream.write(firmado);
494.             sk.getOutputStream().write(outputStream.toByteArray()
   );
495.         }
496.
497.         /* send_msg(msg)
498.          * Metodo interno usado para enviar mensajes en el
   protocolo cifrados y firmados con RSA
499.          * @param: msg, Mensaje sin cifrar siguiendo la
   estructura de mensajes del protocolo
500.          * @param: key, array de bytes que queremos enviar con el
   mensaje
501.          * */
502.         private void send_msg(String msg, byte[] key) throws Exce
   ption {
503.             ByteArrayOutputStream outputStream0 = new ByteArrayOu
   tputStream();
504.             outputStream0.write(key);
505.             outputStream0.write(msg.getBytes());
506.             byte[] enviamos=RSA.encrypt(outputStream0.toByteArray
   ());
507.             byte[] firmado=RSA.firma(enviamos);
508.             ByteArrayOutputStream outputStream = new ByteArrayOut
   putStream();
509.             outputStream.write(enviamos);
510.             outputStream.write(firmado);
511.             sk.getOutputStream().write(outputStream.toByteArray()
   );
512.         }
513.
514.         /* send_msg(msg)
515.          * Metodo interno usado para enviar mensajes en el
   protocolo cifrados con AES
516.          * @param: msg, Mensaje sin cifrar siguiendo la
   estructura de mensajes del protocolo
```

```
517.          * */
518.         private void send_msg_AES(String msg) throws Exception {
519.             byte[] iv=AESCrypto.gernetarekey();
520.             byte[] enviamos=AESCrypto.encrypt(ClientApp.key,iv,ms
   g);
521.             ByteArrayOutputStream outputStream = new ByteArrayOut
   putStream();
522.             outputStream.write(iv);
523.             outputStream.write(enviamos);
524.             sk.getOutputStream().write(outputStream.toByteArray()
   );
525.         }
526.
527.         /* send_msg(msg)
528.          * Metodo interno usado para enviar mensajes en el
   protocolo cifrados con AES
529.          * @param: msg, Mensaje sin cifrar siguiendo la
   estructura de mensajes del protocolo
530.          * @param: token, array de bytes que queremos enviar con
   el mensaje
531.          * */
532.         private void send_msg_AES(String msg,byte[] token) throws
    Exception {
533.             byte[] iv=AESCrypto.gernetarekey();
534.             byte[] enviamos=AESCrypto.encrypt(ClientApp.key,iv,ms
   g);
535.             ByteArrayOutputStream outputStream = new ByteArrayOut
   putStream();
536.             outputStream.write(token);
537.             outputStream.write(iv);
538.             outputStream.write(enviamos);
539.             sk.getOutputStream().write(outputStream.toByteArray()
   );
540.         }
541.
542.
543.         /* make_msg(msg)
544.          * Metodo interno usado para crear mensajes sin cifrar
   siguiendo la estructura de mensajes del protocolo
545.          * @param: ID_mensaje, identificador numerico del mensaje
546.          * @param: datos, conjunto de datos que incluir en el
   mensaje
547.          * */
548.         private String make_msg(int ID_mensaje, String[] datos){
549.             String mensaje=String.valueOf(ID_mensaje);
550.             mensaje+=SEPARADOR_CODIGO;
551.             for (int i=0;i<datos.length;i++){
552.                 mensaje+=datos[i];
553.                 mensaje+=SEPARADOR_DATOS;
554.             }
555.             return mensaje;
556.         }
557.
558.         /* act_SaltyHash()
559.          * Metodo interno con el que se calculan las claves de
   acceso al sistema (ValSM)
560.          * */
561.         private byte[] act_SaltyHash(){
562.             //Realizamos la funcion hash SHA-256 sobre las
   variables concatenadas ID, salt T0 y T1.
```

```
563.             String concatenacion=ID.concat(salt).concat(T0).conca
   t(T1);
564.             return RSA.hash(concatenacion.getBytes());
565.         }
566.
567.         /* read_msg(msg)
568.          * Metodo interno para obtener los datos de un mensaje
   del protocolo
569.          * @param: msg, Mensaje sin cifrar siguiendo la
   estructura de mensajes del protocolo
570.          * */
571.         private String[] read_msg(String msg){
572.             String[] tokens=msg.split(SEPARADOR_CODIGO);
573.             if (tokens.length!=2)
574.                 return null;
575.             String[] mensaje=tokens[1].split(SEPARADOR_DATOS);
576.             return mensaje;
577.         }
578.
579.         /* get_code_msg(msg)
580.          * Metodo interno para obtener el codigo de un mensaje
   del protocolo
581.          * @param: msg, Mensaje sin cifrar siguiendo la
   estructura de mensajes del protocolo
582.          * */
583.         private int get_code_msg(String msg){
584.             String[] tokens=msg.split(SEPARADOR_CODIGO);
585.             if (tokens.length!=2)
586.                 return -1;
587.             return Integer.parseInt(tokens[0]);
588.
589.         }
590.
591.     }
```

Clase AESCrypto.java:
Clase auxiliar que incluye la parte de la criptografía de clave simétrica usada en este trabajo, es decir, el cifrado AES.

```
1. import android.util.Log;
2.
3. import java.security.SecureRandom;
4.
5. import javax.crypto.Cipher;
6. import javax.crypto.spec.IvParameterSpec;
7. import javax.crypto.spec.SecretKeySpec;
8.
9.
10.      /*Clase AESCrypto
11.      * Contiene métodos para realizar el cifrado AES y para
   generar claves AES de 16 bytes
12.      * @author: Omar Molinero Gil
13.      * @version: 15/06/18-v3*/
14.
15.      public class AESCrypto {
16.
```

```
17.          /*generatekey()
18.          * Método usado para generar una clave AES de 16 bytes*/
19.          public static byte[] gernetarekey(){
20.              //Se genera una clave aleatoria usando la clase de
   numeros aleatorios seguros de java
21.              SecureRandom random=new SecureRandom();
22.              byte[] key = new byte[16];
23.              random.nextBytes(key);
24.              return key;
25.          }
26.
27.
28.          /*encrypt(key, initVector,value)
29.          * Método usado para encriptar una cadena de caracteres
   (value), con una clave (key)
30.          * y un vector de inicializacion dado (initVector)
31.          * @param: key, array de bytes de la clave AES
32.          * @param: initVector, array de bytes del IV de AES
33.          * @param: value, String que se quiere encriptar*/
34.          public static byte[] encrypt(byte[] key, byte[] initVecto
   r, String value) {
35.              try {
36.                  //Creamos la clase del vector de inicializacion
37.                  IvParameterSpec
   iv = new IvParameterSpec(initVector);
38.
39.                  //Creamos la clase de la clave AES
40.                  SecretKeySpec
   skeySpec = new SecretKeySpec(key, "AES");
41.
42.                  //Obtenemos la instancia del cifrado AES, modo
   CBC
43.                  Cipher
   cipher = Cipher.getInstance("AES/CBC/PKCS5PADDING");
44.
45.                  //Enciptamos los datos
46.                  cipher.init(Cipher.ENCRYPT_MODE, skeySpec, iv);
47.                  byte[] encrypted = cipher.doFinal(value.getBytes(
   ));
48.                  return encrypted;
49.
50.              } catch (Exception ex) {
51.                  // Mensaje de error
52.                  Log.e("AESCrypto", "Error al encriptar con AES");
53.                  ex.printStackTrace();
54.              }
55.
56.              return null;
57.          }
58.
59.          /*decrypt(key, initVector,encripted)
60.          * Método usado para desencriptar un array de bytes
   (encrypted), con una clave (key)
61.          * y un vector de inicializacion dado (initVector)
62.          * @param: key, array de bytes de la clave AES
63.          * @param: initVector, array de bytes del IV de AES
64.          * @param: encrpted, array de bytes que se quiere
   desencriptar*/
65.          public static String decrypt(byte[] key, byte[] initVecto
   r, byte[] encrypted) {
66.              try {
```

```
67.                 //Creamos la clase del vector de inicializacion
68.                 IvParameterSpec
   iv = new IvParameterSpec(initVector);
69.
70.                 //Creamos la clase de la clave AES
71.                 SecretKeySpec
   skeySpec = new SecretKeySpec(key, "AES");
72.
73.                 //Obtenemos la instancia del cifrado AES, modo
   CBC
74.                 Cipher
   cipher = Cipher.getInstance("AES/CBC/PKCS5PADDING");
75.
76.                 //Desenciptamos los datos
77.                 cipher.init(Cipher.DECRYPT_MODE, skeySpec, iv);
78.                 byte[] original = cipher.doFinal(encrypted);
79.                 return new String(original);
80.
81.             } catch (Exception ex) {
82.                 // Mensaje de error
83.                 Log.e("AESCrypto", "Error al desencriptar con
   AES");
84.                 ex.printStackTrace();
85.             }
86.
87.             return null;
88.         }
89.
90.     }
```

Clase RSACrypto.java:
Clase auxiliar que contiene la criptografía de clave asimétrica usada en este TFG. También incluye métodos para firmar y verificar firmas digitales RSA. Además, está incluido un método que realiza la función hash SHA-256.

```
1. import android.content.Context;
2. import android.support.annotation.Nullable;
3. import android.util.Log;
4.
5. import com.example.root.damas.R;
6.
7. import java.io.InputStream;
8. import java.security.KeyFactory;
9. import java.security.MessageDigest;
10.     import java.security.PrivateKey;
11.     import java.security.PublicKey;
12.     import java.security.Signature;
13.     import java.security.spec.KeySpec;
14.     import java.security.spec.PKCS8EncodedKeySpec;
15.     import java.security.spec.X509EncodedKeySpec;
16.
17.     import javax.crypto.Cipher;
18.
19.     /*Clase RSACrypto
20.     * Contiene métodos para realizar el cifrado RSA, realizar
   firmas RSA y poder verificarlas,
```

```
21.       * cargar claves publica y privada y realizar la funcion hash
   SHA-256
22.       * @author: Omar Molinero Gil
23.       * @version: 10/06/18-v7*/
24.
25.      public class RSACrypto {
26.
27.          /*Variable que guarda el contexto de la aplicacion*/
28.          private Context c;
29.
30.          /*Variable que guarda el cifrado RSA*/
31.          private Cipher rsa;
32.
33.          /*Variable que guarda la clave publica usada para
   encriptar mensajes*/
34.          private PublicKey publicKey;
35.
36.          /*Variable que guarda la clave privada usada para
   desencriptar mensajes*/
37.          private PrivateKey privateKey;
38.
39.          /*RSACrypto(c,flag)
40.          * Constructor de la clase.
41.          * @param: c, Contexto para  poder cargar los datos
   correctamente
42.          * @param: flag, bandera usada para indicar las claves
   usadas en el cifrado
43.          * */
44.          public RSACrypto(Context c, int flag) {
45.              this.c=c;
46.              try {
47.
48.                  // Se carga la clave publica
49.                  publicKey = loadPublicKey(flag);
50.
51.                  // Se carga la clave privada
52.                  privateKey = loadPrivateKey();
53.
54.                  // Obtenemos la clase para encriptar y
   desencriptar
55.                  rsa = Cipher.getInstance("RSA/ECB/PKCS1Padding");
56.              }catch (Exception ex)
57.              {
58.                  // Mensaje de error
59.                  Log.e("RSACrypto", "Error al leer las claves");
60.              }
61.          }
62.
63.          /*encrypt(text)
64.          * Metodo usado para encriptar datos usando el cifrado RSA
65.          * @param: text, Texto a cifrar, en array de bytes
66.          * */
67.          public byte[] encrypt(byte[] text) throws Exception {
68.              // Encriptamos con la clave publica
69.              try {
70.                  rsa.init(Cipher.ENCRYPT_MODE, publicKey);
71.                  return rsa.doFinal(text);
72.              } catch (Exception ex){
73.                  // Mensaje de error
74.                  Log.e("RSACrypto", "Error al
   encriptar" + ex.getMessage());
```

```
75.             }
76.             return null;
77.         }
78.
79.
80.
81.         /*decrypt(text)
82.         * Metodo usado para desencriptar datos mediante el
    algoritmo RSA.
83.         * @param: text, Texto a descifrar, en array de bytes
84.         * */
85.         public byte[] decrypt(byte[] text) throws Exception {
86.             // Desencriptamos con la clave privada
87.             try{
88.             rsa.init(Cipher.DECRYPT_MODE, privateKey);
89.             byte[] bytesDesencriptados = rsa.doFinal(text);
90.             return bytesDesencriptados;
91.             }catch (Exception ex){
92.                 // Mensaje de error
93.                 Log.e("RSACrypto", "Error al
    desencriptar"+ ex.getMessage());
94.             }
95.             return null;
96.         }
97.
98.         /*loadPublicKey(flag)
99.         * Metodo interno usado para cargar claves publicas
100.        * @param: flag, bandera que indica la clave publica a
    cargar
101.        * */
102.        @Nullable
103.        private PublicKey loadPublicKey(int flag) throws Exceptio
    n {
104.            try {
105.                InputStream fraw;
106.
107.                //Cargamos la clave publica dependiendo de la
    bandera
108.                if (flag==ClientApp.SU_PUBLIC){
109.                    fraw =  c.getResources().openRawResource(R.ra
    w.supublic);
110.
111.                }else if (flag==ClientApp.M_PUBLIC){
112.                    fraw =  c.getResources().openRawResource(R.ra
    w.mpublic);
113.
114.                }else if (flag==ClientApp.SM_PUBLIC){
115.                    fraw =  c.getResources().openRawResource(R.ra
    w.smpublic);
116.
117.                } else {
118.                    fraw = c.getResources().openRawResource(R.raw
    .publickey);
119.                }
120.
121.                // Convertimos los datos leidos en un array de
    bytes
122.                int size=fraw.available();
123.                byte[] key=new byte[size];
124.                fraw.read(key);
125.                fraw.close();
```

```
126.
127.                 //Devolvermos la clave publica en su formato
   correcto
128.                 // para que esta pueda ser usada por el algoritmo
   RSA
129.                 KeyFactory keyFactory = KeyFactory.getInstance("R
   SA");
130.                 KeySpec keySpec = new X509EncodedKeySpec(key);
131.                 PublicKey keyFromBytes = keyFactory.generatePubli
   c(keySpec);
132.                 return keyFromBytes;
133.             }
134.             catch (Exception ex) {
135.                 // Mensaje de error
136.                 Log.e("RSACrypto", "Error al cargar la clave
   publica: " + ex.getMessage());
137.             }
138.             return null;
139.
140.         }
141.
142.         /*privateKey()
143.         * Metodo interno usado para cargar la clave privada de Ui
144.         * */
145.         @Nullable
146.         private PrivateKey loadPrivateKey() throws Exception {
147.             try {
148.                 //Cargamos la clave privada
149.                 InputStream fraw = c.getResources().openRawResour
   ce(R.raw.privatekey);
150.
151.                 // Convertimos los datos leidos en un array de
   bytes
152.                 int size=fraw.available();
153.                 byte[] key=new byte[size];
154.                 fraw.read(key);
155.                 fraw.close();
156.
157.                 //Devolvermos la clave privada en su formato
   correcto
158.                 // para que esta pueda ser usada por el algoritmo
   RSA
159.                 KeyFactory keyFactory = KeyFactory.getInstance("R
   SA");
160.                 KeySpec keySpec = new PKCS8EncodedKeySpec(key);
161.                 PrivateKey keyFromBytes = keyFactory.generatePriv
   ate(keySpec);
162.                 return keyFromBytes;
163.             }
164.             catch (Exception ex) {
165.                 // Mensaje de error
166.                 Log.e("RSACrypto", "Error al cargar la clave
   privada: " + ex.getMessage());
167.             }
168.             return null;
169.         }
170.
171.         /*firma(datos)
172.         * Metodo usado para firmar datos
173.         * @param: datos, array de bytes que se quieren firmar
174.         * */
```

```
175.         public byte[] firma(byte[] datos){
176.             try {
177.                 //Usamos una firma RSA al resumen SHA256 de los
    datos introducidos
178.                 Signature firma = Signature.getInstance("SHA256wi
    thRSA");
179.                 firma.initSign(privateKey);
180.                 firma.update(datos);
181.                 return firma.sign();
182.             } catch(Exception ex) {
183.                 // Mensaje de error
184.                 Log.e("FIRMA", "Error al firmar
    " + ex.getMessage());
185.             }
186.             return null;
187.         }
188.
189.         /*verify(datos,datosfirmados)
190.         * Metodo usado para verificar si los datos han sido
    firmados correctamente
191.         * @param: datos, array de bytes que se quiere comprobar
192.         * @param: datosfirmados, firma rsa del array de bytes
    anterior
193.         * */
194.         public boolean verify(byte[] datos,byte[] datosfirmados){
195.             try {
196.                 //Verificamos que datos firmados son en realidad
    la firma RSA
197.                 //del resumen SHA256 de los datos introducidos
198.                 Signature firma = Signature.getInstance("SHA256wi
    thRSA");
199.                 firma.initVerify(publicKey);
200.                 firma.update(datos);
201.                 return firma.verify(datosfirmados);
202.             } catch(Exception ex) {
203.                 // Mensaje de error
204.                 Log.e("FIRMA", "Error al verificar
    " + ex.getMessage());
205.             }
206.             return false;
207.         }
208.
209.         /*hash(datos)
210.         * Metodo usado para realizar la funcion hash SHA-256
211.         * @param: datos, array de bytes de entrada
212.         * */
213.         public byte[] hash(byte[] datos){
214.             try {
215.                 // Calculamos la funcion SHA-256 de los datos
    introducidos
216.                 MessageDigest digest = MessageDigest.getInstance(
    "SHA-256");
217.                 return digest.digest(datos);
218.             } catch(Exception ex) {
219.                 // Mensaje de error
220.                 Log.e("HASH", "Error en la funcion hash
    " + ex.getMessage());
221.             }
222.             return null;
223.         }
224.
```

```
225.     }
```

Python

La parte de Python se encuentra dividida de una manera diferente a la de Java. Cada actor (SU, M y SM) tienen su propio fichero en el que realizan todas sus acciones durante el protocolo. También se incluyen otros ficheros Python con las tecnologías de la información usadas en este trabajo.

SU.py:
Realiza la funcionalidad del actor SU en el protocolo.

```
1. #Actor SU
2. #16/06/18
3.
4. import socket
5. import sys
6. import threading
7. import RSA
8. import AES
9. import DB
10.      import Utils
11.
12.
13.      #Esta funcion realiza el intercambio de mensajes de SU a
   partir de que Ui le envia el
14.      #primer mensaje de la fase 1 del protocolo
15.      def read_data(data):
16.          #Comprobamos firma
17.          if (RSA.check_firma(data[:128],data[128:],Utils.PUBLIC_KE
   Y)==False):
18.              b=RSA.encrypt(Utils.make_msg_error("ERROR EN LA
   FIRMA"),Utils.PUBLIC_KEY)
19.              f=RSA.firma(b,Utils.SU_PRIVATE_KEY)
20.              connection.send(b)
21.              connection.send(f)
22.              print('ERROR AL COMPROBAR LA FIRMA')
23.              return
24.
25.          print('Firma recibida correcta')
26.          #desciframos el mensaje, que se encuentra en los primeros
   128 bytes
27.          msg=str(RSA.decrypt(data[:128],Utils.SU_PRIVATE_KEY))
28.
29.          #Imprimimos el mensaje recibido
30.          print('Mensaje recibido de Ui: '+ msg)
31.
32.          #Obtenemos los datos del mensaje recibido
33.          code, mensaje=Utils.read_msg(msg)
34.          if (int(code)!=Utils.P_REGISTRO):
35.              b=RSA.encrypt(Utils.make_msg_error("ERROR AL LEER LOS
   DATOS"),Utils.PUBLIC_KEY)
36.              f=RSA.firma(b,Utils.SU_PRIVATE_KEY)
37.              connection.send(b)
38.              connection.send(f)
```

```
39.                 print('ERROR EN EL CONTENIDO DEL MENSAJE RECIBIDO')
40.                 return
41.             #Comprueba que el cliente es valido accediendo a una base
    de datos. Si no lo es, mensaje de error.
42.             if (DB.check_is_valid_client(mensaje[0],mensaje[1])==Fals
    e):
43.                 b=RSA.encrypt(make_msg_error("ERROR EN EL
    REGISTRO"),Utils.PUBLIC_KEY)
44.                 f=RSA.firma(b,Utils.SU_PRIVATE_KEY)
45.                 connection.send(b)
46.                 connection.send(f)
47.                 print('CLIENTE NO VALIDO PARA REGISTRARSE')
48.                 return
49.
50.             print('Nos conectamos con M')
51.
52.             #Generamos un ID aleatorio de 128 bits
53.             random_ID=RSA.get_random_ID()
54.
55.             #crea una conexion con M.
56.             s=socket.socket(socket.AF_INET,socket.SOCK_STREAM)
57.             s.connect(('localhost',9000))
58.
59.             print('Enviamos mensaje del protocolo a M')
60.             env=RSA.encrypt(Utils.make_msg_agreement(random_ID),Utils
    .M_PUBLIC_KEY)
61.             fenv=RSA.firma(env,Utils.SU_PRIVATE_KEY)
62.             s.send(env)
63.             s.send(fenv)
64.
65.             #Espera la respuesta de M
66.             r=s.recv(128)
67.             r2=s.recv(128)
68.             print('Mensaje recibido de M:
    '+str(RSA.decrypt(r,Utils.SU_PRIVATE_KEY)))
69.             code, mensajee = Utils.read_msg(str(RSA.decrypt(r,Utils.S
    U_PRIVATE_KEY)))
70.
71.             print('Enviamos el mensaje de respuesta a Ui')
72.             b=RSA.encrypt(Utils.make_msg_response_ok(str(random_ID),m
    ensajee[0],mensajee[1]),Utils.PUBLIC_KEY)
73.             f=RSA.firma(b,Utils.SU_PRIVATE_KEY)
74.             connection.send(b)
75.             connection.send(f)
76.
77.             DB.save_database_SU(str(random_ID),mensaje[0],DB.BLOCKCHA
    IN_SU)
78.
79.         #Esta funcion realiza la funcion de SU en la fase 4 del
    protocolo
80.         def read_data2(data):
81.                 print('Fase 4 del protocolo criptografico')
82.
83.                 #desciframos el mensaje
84.                 msg=str(RSA.decrypt(data[:128],Utils.SU_PRIVATE_KEY))
85.                 print("Hemos recibido de SM: "+msg)
86.                 #Obtenemos los datos del mensaje recibido
87.                 code, mensaje=Utils.read_msg(msg)
88.
89.             #leemos de BC de SU la identidad real del usuario
90.                 DNI=DB.get_DNI_Blockchain_SU(mensaje[0])
```

```
91.
92.          #se la enviamos a SM cifrada y firmada
93.              env=RSA.encrypt(Utils.make_msg_revocacion_res_SU(DNI)
   ,Utils.SM_PUBLIC_KEY)
94.              fenv=RSA.firma(env,Utils.SU_PRIVATE_KEY)
95.              connection.send(env)
96.              connection.send(fenv)
97.
98.
99.      def thread_connection(connection,client_address):
100.         try:
101.             while True:
102.                 print('Leemos el mensaje recibido')
103.                 data = connection.recv(256)
104.                 if data:
105.                     #Si el origen del mensaje es el SM nos esta
   realizando una peticion de revocacion de identidad
106.                     #por lo que actuamos de manera diferente
107.                     if DB.is_SM(client_address[1]):
108.                             read_data2(data)
109.                             break
110.
111.                     read_data(data)
112.                 else:
113.                     break
114.         finally:
115.             print('Cerramos la conexion con el cliente')
116.             connection.close()
117.
118.
119.     # Creando el socket TCP/IP
120.     sock = socket.socket(socket.AF_INET, socket.SOCK_STREAM)
121.     # Enlace de socket y puerto
122.     server_address = ('localhost', 10000)
123.     print ('Levantamos el servidor SU. IP: localhost, puerto:
   10000')
124.     sock.bind(server_address)
125.     # Escuchando conexiones entrantes
126.     sock.listen(1)
127.
128.     while True:
129.         # Esperando conexion
130.         print ('Esperando para conectarse')
131.         connection, client_address = sock.accept()
132.         print ('Se ha establecido una conexion con un actor')
133.         t=threading.Thread(target=thread_connection, args=(connec
   tion,client_address))
134.         t.start()
135.
```

M.py:
Realiza la funcionalidad del actor M en el protocolo.

```
1. #Actor M
2. #16/06/18
3.
4. import socket
```

```
5. import sys
6. import threading
7. import RSA
8. import AES
9. import Utils
10.      import DB
11.      import time
12.      import random
13.      import string
14.
15.      #Esta funcion realiza el intercambio de informacion de M
   cuando recibe un mensaje de SU
16.      #Es decir, esto se encuentra dentro de la fase 1 del
   protocolo
17.      def read_data(data,data2):
18.          #Comprobamos firma
19.          if (RSA.check_firma(data,data2, Utils.SU_PUBLIC_KEY)==Fal
   se):
20.              print("Firma recibida falsa")
21.              return
22.          #desciframos el mensaje, que se encuentra en los primeros
   128 bytes
23.          msg=str(RSA.decrypt(data,Utils.M_PRIVATE_KEY))
24.          print('Mensaje de SU recibido: ' + msg)
25.
26.          #Obtenemos los datos del mensaje
27.          code, mensaje=Utils.read_msg(msg)
28.          if (int(code)!=Utils.P_ACUERDO_M):
29.              b=RSA.encrypt(Utils.make_msg_error("ERROR AL LEER LOS
   DATOS"),Utils.SU_PUBLIC_KEY)
30.              f=RSA.firma(b,Utils.M_PRIVATE_KEY)
31.              connection.send(b)
32.              connection.send(f)
33.              return
34.          #Genera seed y val y timestamp
35.          ts=(int)(time.time())
36.          seed=random.getrandbits(16)
37.          valM_0 = ''.join([random.choice(string.ascii_letters + st
   ring.digits) for n in range(32)])
38.
39.          #Envia el mensaje a SU
40.          print('Enviamos el mensaje de respuesta a SU')
41.          b=RSA.encrypt(Utils.make_msg_response_valm(str(ts),valM_0
   ),Utils.SU_PUBLIC_KEY)
42.          f=RSA.firma(b,Utils.M_PRIVATE_KEY)
43.          connection.send(b)
44.          connection.send(f)
45.
46.          #Guardamos la informacion necesaria en las bases de datos
47.          DB.save_data(mensaje[0],str(ts),DB.DATABASE_M_CLIENTS)
48.          DB.save_token(DB.DATABASE_M_TOKENS,valM_0)
49.
50.
51.      #Esta funcion realiza el intercambio de mensajes cuando M
   recibe un mensaje de Ui
52.      #Es decir, la fase 2 del protocolo
53.      def read_data2(data):
54.              if(RSA.check_firma(data[:128],data[128:],Utils.PUBLIC
   _KEY)==False):
55.                      print("Firma recibida falsa")
56.                      return
```

```
57.                 msg=str(RSA.decrypt(data[:128],Utils.M_PRIVATE_KEY)[1
   6:])
58.                 print('Mensaje de Ui recibido: '+msg)
59.                 code, mensaje=Utils.read_msg(msg)
60.                 #Comprobamos que el token de validacion es correcto y
   lo eliminamos mensaje[0]
61.                 if(not DB.check_valm(mensaje[0])):
62.                         print("ValM no encontrado")
63.                         return
64.                 print("ValM correcto")
65.
66.                 #Comprobamos que el ID esta de entre la lista de
   clientes y guardamos su Tinicial mensaje[2]
67.                 T0=DB.read_T0(mensaje[2])
68.                 if T0 is None:
69.                         print("Cliente no encontrado")
70.                         return
71.                 print("Cliente correcto")
72.                 ts=time.time()
73.
74.                 #Enviamos mensaje con T1 y el nuevo ts cifrado con
   clave compartida
75.                 key=RSA.decrypt(data[:128],Utils.M_PRIVATE_KEY)[:16]
76.                 connection.send(AES.AESCipher(key).encrypt(Utils.make
   _msg_access(mensaje[1],ts)))
77.                 data2=connection.recv(128)
78.
79.                 msg2=AES.AESCipher(key).decrypt(data2[0:16],data2[16:
   ])
80.                 print("Recibimos el segundo mensaje de Ui" +msg2)
81.                 code2, mensaje2=Utils.read_msg_AES(msg2)
82.
83.
84.                 concatenacion=""+mensaje[2]+""+mensaje2[0]+""+T0+""+s
   tr(ts)
85.
86.                 client_hash=RSA.get_hash(str.encode(concatenacion)).d
   igest()
87.
88.                 print('Generamos la clave de acceso al sistema
   (ValSM) y la guardamos en la BD que controla SM')
89.                 DB.save_new_access(DB.DATABASE_SM_TOKENS,client_hash)
90.                 DB.save_m_blockchain(DB.BLOCKCHAIN_M,client_hash,mens
   aje[2])
91.
92.                 print('Enviamos un mensaje a Ui dando por finalizada
   la fase 2 del protocolo')
93.                 connection.send(AES.AESCipher(key).encrypt(Utils.make
   _msg_access_ok()))
94.
95.
96.         def read_data3(data):
97.                 msg=str(RSA.decrypt(data,Utils.M_PRIVATE_KEY))
98.                 print('Mensaje de SM recibido: '+msg)
99.                 code, mensaje=Utils.read_msg(msg)
100.                #Si SM nos pide un nuevo token, estamos en el
   protocolo 3
101.                if (int(code)==Utils.P_PETICION_TOKEN_M):
102.                        print('Protocolo 3. Peticion de token')
103.                        #Creamos un nuevo token de acceso, lo
   guardamos y se lo enviamos a SM
```

```
104.                     valM_1 = ''.join([random.choice(string.ascii_
   letters + string.digits) for n in range(32)])
105.                     DB.save_token(DB.DATABASE_M_TOKENS,valM_1)
106.                     b=RSA.encrypt(Utils.make_msg_respuesta_val_M(
   valM_1),Utils.SM_PUBLIC_KEY)
107.                     f=RSA.firma(b,Utils.M_PRIVATE_KEY)
108.                     connection.send(b)
109.                     connection.send(f)
110.                     print('Enviamos el token a SM')
111.             #si no, estamos en el protocolo 4
112.             else:
113.                     print('Protocolo 4. Revocacion de anonimia')
114.                     #recuperamos el ID del usuario
115.                     ID=DB.get_ID_Blockchain_M(mensaje[0])
116.                     #Se lo enviamos a SM
117.                     b=RSA.encrypt(Utils.make_msg_revocacion_res_M
   (ID),Utils.SM_PUBLIC_KEY)
118.                     f=RSA.firma(b,Utils.M_PRIVATE_KEY)
119.                     connection.send(b)
120.                     connection.send(f)
121.                     print('Enviamos el token a SM')
122.
123.
124.
125.     def thread_connection(connection,client_address):
126.         try:
127.             #Caso en el que el origen del mensaje sea el actor SM
128.             if (DB.is_SM(client_address[1])==True):
129.                     print('SM ha establecido conexion con
   nosotros')
130.                     data=connection.recv(128)
131.                     read_data3(data)
132.                     return;
133.
134.             #Caso en el que el origen del mensaje sea el actor Ui
135.             elif (DB.is_SU(client_address[1])==False):
136.                     print('Fase 2 del protocolo')
137.                     print('Ui ha establecido conexion con
   nosotros')
138.                     data=connection.recv(256)
139.                     read_data2(data)
140.                     connection.close()
141.                     return;
142.
143.
144.             #Caso en el que el origen del mensaje sea el actor SU
145.             print('SU ha establecido conexion con nosotros')
146.             data = connection.recv(128)
147.             data2= connection.recv(128)
148.             if data and data2:
149.                 read_data(data,data2)
150.             else:
151.                     print('No hemos recibido nada')
152.         finally:
153.             print('Cerramos la conexion con el cliente')
154.             connection.close()
155.
156.
157.     # Creando el socket TCP/IP
158.     sock = socket.socket(socket.AF_INET, socket.SOCK_STREAM)
159.     # Enlace de socket y puerto
```

```
160.     server_address = ('localhost', 9000)
161.     print ('Levantamos el servidor M. IP: localhost, puerto:
   9000')
162.     sock.bind(server_address)
163.     # Escuchando conexiones entrantes
164.     sock.listen(1)
165.
166.     while True:
167.         # Esperando conexion
168.         print ('Esperando para conectarse')
169.         connection, client_address = sock.accept()
170.         print ('Se ha establecido una conexion con un actor')
171.         t=threading.Thread(target=thread_connection, args=(connec
   tion,client_address))
172.         t.start()
```

SM.py:
Realiza la funcionalidad del actor SM en el protocolo.

```
1. #Actor M
2. #19/06/18
3.
4. import socket
5. import sys
6. import threading
7. import RSA
8. import AES
9. import Utils
10.      import DB
11.      import time
12.      import random
13.      import string
14.
15.
16.      #Realiza la funcionalidad de las fases 3 y 4 del protocolo
17.      #dependiendo del resultado de check_integrity
18.      def read_data(data):
19.              print("Fase 3 del protocolo")
20.              if(RSA.check_firma(data[:128],data[128:],Utils.PUBLIC
   _KEY)==False):
21.                      print("Firma falsa")
22.                      return
23.
24.              #desciframos el mensaje recibido
25.              msg=str(RSA.decrypt(data[:128],Utils.SM_PRIVATE_KEY)[
   16:])
26.              print('Mensaje de Ui recibido: '+msg)
27.              code, mensaje=Utils.read_msg(msg)
28.              #Enviamos un mensaje con la marca temporar antigua y
   la nueva
29.              ts=time.time()
30.              #Obtenemos la clave de cifrado AES
31.              key=RSA.decrypt(data[:128],Utils.SM_PRIVATE_KEY)[:16]
32.              #Enviamos la respuesta a Ui cifrada con AES
33.              connection.send(AES.AESCipher(key).encrypt(Utils.make
   _msg_access_SM(mensaje[0],ts)))
34.
```

```
35.                 #Esperamos la respuesta de Ui
36.                 data2=connection.recv(128)
37.                 ValSM=data2
38.
39.                 if(not DB.check_token_sm(data2)):
40.                         print("La clave de acceso recibida no es
   valida. EL cliente no tiene acceso al sistema")
41.                         connection.send(AES.AESCipher(key).encrypt(Ut
   ils.make_msg_access_SM_err("Clave de acceso invalida")))
42.                 else:
43.                         print("Clave de acceso valida. El usuario
   anonimo tiene acceso a nuestro servicio")
44.
45.                 #enviamos mensaje de acceso correcto
46.                 connection.send(AES.AESCipher(key).encrypt(Utils.make
   _msg_access_SM_ok("Acceso correcto")))
47.
48.
49.                 #Recibimos peticion de logout
50.                 data2=connection.recv(128)
51.                 msg2=AES.AESCipher(key).decrypt(data2[0:16],data2[16:
   ])
52.                 print("Recibimos el mensaje de Ui: " +msg2)
53.                 code2, mensaje2=Utils.read_msg_AES(msg2)
54.
55.                 #comprobamos si el usuario ha incumplido el contrato.
56.                 #La variable valor indicara que lo ha hecho si es
   false, true si ha actuado correctamente
57.                 valor, prueba=DB.check_integrity()
58.                 if (valor==True):
59.                         #En este caso el usuario ha actuado
   correctamente. Se debe seguir con el protocolo 3
60.                         print('El usuario no ha cometido ninguna
   ilegalidad en su sesion. Procedemos con el final del protocolo 3')
61.                         #crea una conexion con M.
62.                         s=socket.socket(socket.AF_INET,socket.SOCK_ST
   REAM)
63.                         s.connect(('localhost',9000))
64.
65.                         print('Enviamos mensaje del protocolo a M')
66.                         env=RSA.encrypt(Utils.make_msg_solicitud_val_
   M('Mensaje de peticion de token'),Utils.M_PUBLIC_KEY)
67.                         fenv=RSA.firma(env,Utils.SM_PRIVATE_KEY)
68.                         s.send(env)
69.                         s.send(fenv)
70.
71.                         #Espera la respuesta de M
72.                         r=s.recv(128)
73.                         r2=s.recv(128)
74.                         print('Mensaje recibido de M:
   '+str(RSA.decrypt(r,Utils.SM_PRIVATE_KEY)))
75.                         code, mensajee = Utils.read_msg(str(RSA.decry
   pt(r,Utils.SM_PRIVATE_KEY)))
76.
77.                         #Enviamos al usuario el nuevo token de M
78.                         connection.send(AES.AESCipher(key).encrypt(Ut
   ils.make_msg_access_SM_ok(mensajee[0])))
79.                 else:
80.                         #En este caso el usuario ha cometido alguna
   infraccion. Comienza el protocolo 4
```

```
81.                         print('El usuario ha cometido ninguna al
   menos una ilegalidad en su sesion. Empieza el protocolo 4')
82.
83.                         #Se crea una conexion con M
84.                         s=socket.socket(socket.AF_INET,socket.SOCK_ST
   REAM)
85.                         s.connect(('localhost',9000))
86.
87.                         print('Enviamos mensaje del protocolo a M')
88.                         env=RSA.encrypt(Utils.make_msg_revocacion_M(V
   alSM,prueba),Utils.M_PUBLIC_KEY)
89.                         fenv=RSA.firma(env,Utils.SM_PRIVATE_KEY)
90.                         s.send(env)
91.                         #s.send(fenv)
92.
93.                         #Espera la respuesta de M
94.                         r=s.recv(128)
95.                         r2=s.recv(128)
96.                         print('Mensaje recibido de M:
   '+str(RSA.decrypt(r,Utils.SM_PRIVATE_KEY)))
97.                         code, mensajee = Utils.read_msg(str(RSA.decry
   pt(r,Utils.SM_PRIVATE_KEY)))
98.
99.                         #Se crea una conexion con SU
100.                        s=socket.socket(socket.AF_INET,socket.SOCK_ST
   REAM)
101.                        s.connect(('localhost',10000))
102.
103.                        print('Enviamos mensaje del protocolo a SU')
104.                        env=RSA.encrypt(Utils.make_msg_revocacion_SU(
   mensajee[0],prueba),Utils.SU_PUBLIC_KEY)
105.                        fenv=RSA.firma(env,Utils.SM_PRIVATE_KEY)
106.                        s.send(env)
107.                        #s.send(fenv)
108.
109.                        #Espera la respuesta de M
110.                        r=s.recv(128)
111.                        r2=s.recv(128)
112.                        print('Mensaje recibido de SU:
   '+str(RSA.decrypt(r,Utils.SM_PRIVATE_KEY)))
113.                        code2, mensajee2 = Utils.read_msg(str(RSA.dec
   rypt(r,Utils.SM_PRIVATE_KEY)))
114.
115.                        connection.send(AES.AESCipher(key).encrypt(Ut
   ils.make_msg_revocacion_OK(mensajee2[0],prueba)))
116.
117.
118.    def thread_connection(connection,client_address):
119.        try:
120.            print('ESTAMOS DENTRO')
121.            data = connection.recv(256)
122.
123.            if data:
124.                read_data(data)
125.            else:
126.                    print('No hemos recibido datos')
127.        finally:
128.            print('Cerramos la conexion')
129.            connection.close()
130.
131.
```

```
132.      # Creando el socket TCP/IP
133.      sock = socket.socket(socket.AF_INET, socket.SOCK_STREAM)
134.      # Enlace de socket y puerto
135.      server_address = ('localhost', 8000)
136.      print ('Levantamos el servidor SM. IP: localhost, puerto:
   8000')
137.      sock.bind(server_address)
138.      # Escuchando conexiones entrantes
139.      sock.listen(1)
140.
141.      while True:
142.          # Esperando conexion
143.          print ('Esperando para conectarse')
144.          connection, client_address = sock.accept()
145.          print ('Se ha establecido una conexion con un actor')
146.          t=threading.Thread(target=thread_connection, args=(connec
   tion,client_address))
147.          t.start()
```

RSA.py:
En este fichero se incluye la criptografía de clave asimétrica usada en este TFG. También incluye métodos para firmar y verificar firmas digitales RSA. Además, está incluido un método que realiza la función hash SHA-256.

```
1. import Crypto
2. from Crypto.PublicKey import RSA
3. from Crypto import Random
4. from Crypto.Cipher import PKCS1_v1_5 as Cipher_PKCS1
5. from Crypto.Hash import SHA256
6. from Crypto.Signature import PKCS1_v1_5 as Signature_PKCS1
7. from base64 import b64decode
8. import random
9. import ast
10.
11.
12.       def readkeys(clave):
13.           f = open (clave,'rb')
14.           pkey=f.read()
15.           f.close()
16.           return pkey
17.
18.       def get_hash(data):
19.           return SHA256.new(data)
20.
21.       def firma(data,clave):
22.           key=readkeys(clave)
23.           keyR=RSA.importKey(key,'PEM')
24.           firmar=Signature_PKCS1.new(keyR)
25.           return firmar.sign(get_hash(data))
26.
27.       def check_firma(data,firma,clave):
28.           key=readkeys(clave)
29.           keyR=RSA.importKey(key,'PEM')
30.           comprobar=Signature_PKCS1.new(keyR)
31.           jash=get_hash(data)
32.           return comprobar.verify(jash,firma)
33.
```

```
34.
35.      def encrypt(data, clave):
36.          key=readkeys(clave)
37.          keyR=RSA.importKey(key,'PEM')
38.          cipher = Cipher_PKCS1.new(keyR)
39.          cipher_text = cipher.encrypt(str.encode(data))
40.          return cipher_text
41.
42.      def decrypt(data, clave):
43.          key=readkeys(clave)
44.          keyR=RSA.importKey(key,'PEM')
45.          cipher=Cipher_PKCS1.new(keyR)
46.          dmsg=cipher.decrypt(data,cipher.encrypt(str.encode('NOOOO
   ERRORRR')))
47.          return dmsg
48.
49.      def get_random_ID():
50.              return random.getrandbits(128)
```

AES.py:
Este fichero contiene una clase usada para cifrar o descifrar mensajes según el método criptográfico AES.

```
1. from Crypto import Random
2. from Crypto.Cipher import AES
3.
4.
5.
6. BLOCK_SIZE = 16  # Bytes
7. pad = lambda s: s + (BLOCK_SIZE - len(s) % BLOCK_SIZE) * \
8.                 chr(BLOCK_SIZE - len(s) % BLOCK_SIZE)
9. unpad = lambda s: s[:-ord(s[len(s) - 1:])]
10.
11.
12.      class AESCipher:
13.
14.          def __init__(self, key):
15.              self.key=key
16.
17.
18.          def encrypt(self, raw):
19.              raw = pad(raw)
20.              iv = Random.new().read(AES.block_size)
21.              cipher = AES.new(self.key, AES.MODE_CBC, iv)
22.              return iv + cipher.encrypt(raw)
23.
24.          def decrypt(self, iv, enc):
25.              iv=iv
26.              cipher = AES.new(self.key, AES.MODE_CBC, iv)
27.              return unpad(cipher.decrypt(enc)).decode('utf8')
```